



conference

proceedings

Proceedings of the 2021 USENIX Annual Technical Conference

2021 USENIX Annual Technical Conference

July 14–16, 2021

July 14–16, 2021

ISBN 978-1-939133-23-6

Sponsored by



USENIX ATC '21 Sponsors

Silver Sponsor



Bronze Sponsors



General Sponsor



Industry Partners and Media Sponsors

ACM *Queue*
FreeBSD Foundation
No Starch Press

USENIX Supporters

USENIX Patrons

Bloomberg • Facebook • Google
LinkedIn • Microsoft • NetApp • Salesforce

USENIX Benefactors

Amazon • AuriStor • Discernible • IBM
Shopify • Thinkst Canary • Transcend • Two Sigma

USENIX Partner

Top10VPN

Open Access Publishing Partner

PeerJ

USENIX Association

**Proceedings of the
2021 USENIX Annual Technical Conference**

July 14–16, 2021

© 2021 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 978-1-939133-23-6

Conference Organizers

Program Co-Chairs

Irina Calciu, *VMware Research*
Geoff Kuenning, *Harvey Mudd College*

Program Committee

Sangeetha Abdu Jyothi, *University of California, Irvine, and VMware Research*
Reto Achermann, *University of British Columbia*
Nitin Agrawal, *ThoughtSpot*
Amogh Akshintala, *Facebook*
George Amvrosiadis, *Carnegie Mellon University*
Raja Appuswamy, *EURECOM, Sophia Antipolis*
Anirudh Badam, *Microsoft Research*
Antonio Barbalace, *University of Edinburgh*
Frank Bellosa, *Karlsruhe Institute of Technology*
Daniel S. Berger, *Microsoft Research*
Annette Bieniusa, *Technische Universität Kaiserslautern*
Laurent Bindschaedler, *Massachusetts Institute of Technology*
Eleanor Birrell, *Pomona College*
William J. Bolosky, *Microsoft Research*
James Bottomley, *IBM Research*
Sara Bouchenak, *INSA Lyon*
Nathan Bronson, *Rockset*
Mihai Budiu, *VMware Research*
Anton Burtsev, *University of California, Irvine*
Kevin Butler, *University of Florida*
Marco Canini, *KAUST*
Lydia Chen, *Delft University of Technology*
Young-ri Choi, *UNIST (Ulsan National Institute of Science and Technology)*
Byung-Gon Chun, *Seoul National University*
David Cock, *ETH Zurich*
Jon Crowcroft, *University of Cambridge*
Heming Cui, *The University of Hong Kong (HKU)*
Dilma da Silva, *Texas A&M University*
Alex Daglis, *Georgia Institute of Technology*
Tudor David, *Oracle Labs Zurich*
Dave Dice, *Oracle Labs*
Aleksandar Dragojevic, *Microsoft Research Cambridge*
Abhinav Duggal, *Dell-EMC*
Alexandra Fedorova, *University of British Columbia*
Pascal Felber, *University of Neuchatel*
Anja Feldmann, *Max Planck Institute for Software Systems (MPI-SWS)*
Pedro Fonseca, *Purdue University*
Jessie Frazelle, *Oxide Computer*
Jana Giceva, *Technische Universität Munich*
Ionel Gog, *University of California, Berkeley*
Justin Gottschlich, *Intel Labs and University of Pennsylvania*
Tim Harris, *Microsoft*
Michio Honda, *University of Edinburgh*
Yu Hua, *Huazhong University of Science and Technology*
Zsolt Istvan, *IMDEA*
Bill Jannen, *Williams College*
Vasiliki Kalavri, *Boston University*
Anuj Kalia, *Microsoft*

Sanidhya Kashyap, *EPFL*
Aasheesh Kolli, *Google and The Pennsylvania State University*
Michael Kozuch, *Intel Labs*
John Kubiatowicz, *University of California, Berkeley*
Youngjin Kwon, *Korea Advanced Institute of Science and Technology (KAIST)*
Baptiste Lepers, *University of Sydney*
Heiner Litz, *University of California, Santa Cruz*
Yunxin Liu, *Microsoft Research Asia*
Brandon Lucia, *Carnegie Mellon University*
Xiaosong Ma, *Qatar Computing Research Institute*
Jonathan Mace, *Max Planck Institute for Software Systems (MPI-SWS)*
Carlos Maltzahn, *University of California, Santa Cruz*
Virendra Marathe, *Oracle Labs*
Marcelo Martins, *Apple*
Ali Jose Mashtizadeh, *University of Waterloo*
Alexander Merritt, *BedRock Systems*
Michael Mesnier, *Intel*
Ethan Miller, *University of California, Santa Cruz, and Pure Storage*
Changwoo Min, *Virginia Tech*
Adam Morrison, *Tel Aviv University*
Gilles Muller, *INRIA*
Kiran-Kumar Muniswamy-Reddy, *Oracle*
Amy Murphy, *Bruno Kessler Foundation*
Madan Musuvathi, *Microsoft Research*
Onur Mutlu, *ETH Zurich*
Dalit Naor, *The Academic College of Tel Aviv-Yaffo*
Khanh Nguyen, *Texas A&M University*
Cristina Nita-Rotaru, *Northeastern University*
Sam H. Noh, *UNIST (Ulsan National Institute of Science and Technology)*
Amy Ousterhout, *University of California, Berkeley*
Roberto Palmieri, *Lehigh University*
Mangpo Phothilimthana, *Google*
Peter Pietzuch, *Imperial College London*
Dan Ports, *Microsoft Research*
Costin Raiciu, *Politehnica University of Bucharest*
Michael Reiter, *Duke University*
Larry Rudolph, *Two Sigma*
Leonid Ryzhyk, *VMware Research*
Mahadev Satyanarayanan, *Carnegie Mellon University*
Jiri Schindler, *Tranquil Data*
Eric Schkfuza, *Amazon*
Russell Sears, *Apple*
Sangeetha Seshadri, *IBM Research - Almaden*
Liuba Shrira, *Brandeis University*
Mark Silberstein, *Technion—Israel Institute of Technology*
Michael Spear, *Lehigh University*
Scott Stoller, *Stony Brook University*
Patrick Stuedi, *LinkedIn*
Ryan Stutsman, *University of Utah*
Swaminathan Sundararaman, *Fusion IO*
Steve Swanson, *University of California, San Diego*

Michael Swift, *University of Wisconsin—Madison*
Adriana Szekeres, *VMware Research*
Amy Tai, *VMware Research*
Vasily Tarasov, *IBM Research - Almaden*
Eno Thereska, *Amazon*
Vasileios Trigonakis, *Oracle Labs Zurich*
Dan Tsafir, *Technion—Israel Institute of Technology*
Ymir Vigfusson, *Emory University*
Nandita Vijaykumar, *University of Toronto*
Haris Volos, *University of Cyprus*
Keval Vora, *Simon Fraser University*
Roger Wattenhofer, *ETH Zurich*
Hakim Weatherspoon, *Cornell University*
Ric Wheeler, *Facebook*
Avani Wildani, *Emory University*
John Wilkes, *Google*
Youjip Won, *Korea Advanced Institute of Science and Technology (KAIST)*
Gerd Zellweger, *VMware Research*
Lin Zhong, *Yale University*

Submissions Co-Chairs
Alex Conway, *VMware Research*
Chris Stone, *Harvey Mudd College*

Preview Session Co-Chairs
Sangeetha Abdu Jyothi, *University of California, Irvine, and VMware Research*
Deniz Altinbüken, *Google*
Dilma Da Silva, *Texas A&M University*
Aurojit Panda, *New York University*

Mentoring Co-Chairs
Baris Kasikci, *University of Michigan*
Amy Ousterhout, *University of California, Berkeley*
Malte Schwarzkopf, *Brown University*

Networking Session Co-Chairs
Reto Achermann, *University of British Columbia*
Zsolt István, *IT University of Copenhagen*
Adriana Szekeres, *VMware Research*
Vasily Tarasov, *IBM Research - Almaden*

Awards Committee
David Cock, *ETH Zurich*
Dilma Da Silva, *Texas A&M University*
Michio Honda, *University of Edinburgh*
Yu Hua, *Huazhong University of Science and Technology*
Liuba Shriram, *Brandeis University*

External Reviewers

Ahmed M. Abdelmoniem
Ittai Abraham
Nadav Amit
Shai Bergman
Muhammad Bilal
Chandranil Chakrabortii
Joanna Che

Vijay Chidambaram
Suxia Cui
Haggai Eran
Jiawei Fei
Rajiv Gupta
Kasra Jamshidi
Ming Liu

Ali Najafi
Meni Orenbach
Alon Rashelbach
Lynus Vaz
Minghao Xie
Lior Zeno

2021 USENIX Annual Technical Conference

July 14–16, 2021

Message from the USENIX ATC '21 Program Co-Chairs x

Wednesday, July 14

Peeking over the Fence: RDMA

Naos: Serialization-free RDMA networking in Java 1
Konstantin Taranov, *ETH Zurich*; Rodrigo Bruno, *INESC-ID / Técnico, ULisboa*; Gustavo Alonso and Torsten Hoefler, *ETH Zurich*

One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory 15
Pengfei Zuo, Jiazhao Sun, Liu Yang, and Shuangwu Zhang, *Huawei Cloud*; Yu Hua, *Huazhong University of Science and Technology*

Characterizing and Optimizing Remote Persistent Memory with RDMA and NVM 31
Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang, *Shanghai Jiao Tong University; Shanghai AI Laboratory; Engineering Research Center for Domain-specific Operating Systems*

MigrOS: Transparent Live-Migration Support for Containerised RDMA Applications 47
Maksym Planeta and Jan Bierbaum, *TU Dresden*; Leo Sahaya Daphne Antony, *AMOLF*; Torsten Hoefler, *ETH Zürich*; Hermann Härtig, *TU Dresden*

Dogs Never Get Tired: Power and Edge Computing

Prediction-Based Power Oversubscription in Cloud Platforms 65
Alok Gautam Kumbhare, Reza Azimi, Ioannis Manousakis, Anand Bonde, Felipe Frujeri, Nithish Mahalingam, Pulkit A. Misra, Seyyed Ahmad Javadi, Bianca Schroeder, Marcus Fontoura, and Ricardo Bianchini, *Microsoft Research and Microsoft Azure*

Proactive Energy-Aware Adaptive Video Streaming on Mobile Devices 81
Jiayi Meng, Qiang Xu, and Y. Charlie Hu, *Purdue University*

Video Analytics with Zero-streaming Cameras 99
Mengwei Xu, *Peking University/Beijing University of Posts and Telecommunications*; Tiantu Xu, *Purdue ECE*; Yunxin Liu, *Institute for AI Industry Research (AIR), Tsinghua University*; Felix Xiaozhu Lin, *University of Virginia*

ASAP: Fast Mobile Application Switch via Adaptive Prepaging 117
Sam Son, Seung Yul Lee, Yunho Jin, and Jonghyun Bae, *Seoul National University*; Jinkyu Jeong, *Sungkyunkwan University*; Tae Jun Ham and Jae W. Lee, *Seoul National University*; Hongil Yoon, *Google*

Barking up the Wrong Tree: Correctness and Debugging

PyLIVE: On-the-Fly Code Change for Python-based Online Services 131
Haochen Huang, Chengcheng Xiang, Li Zhong, and Yuanyuan Zhou, *University of California, San Diego*

RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing 147
Mingzhe Wang, Jie Liang, Chijin Zhou, and Yu Jiang, *Tsinghua University*; Rui Wang, *Capital Normal University*; Chengnian Sun, *Waterloo University*; Jiaguang Sun, *Tsinghua University*

TCP-Fuzz: Detecting Memory and Semantic Bugs in TCP Stacks with Fuzzing 161
Yong-Hao Zou and Jia-Ju Bai, *Tsinghua University*; Jielong Zhou, Jianfeng Tan, and Chenggang Qin, *Ant Group*; Shi-Min Hu, *Tsinghua University*

MLEE: Effective Detection of Memory Leaks on Early-Exit Paths in OS Kernels 177
Wenwen Wang, *University of Georgia*

Argus: Debugging Performance Issues in Modern Desktop Applications with Annotated Causal Tracing 193
Lingmei Weng, *Columbia University*; Peng Huang, *Johns Hopkins University*; Jason Nieh and Junfeng Yang, *Columbia University*

Searching for Tracks: Graphs

- aDFS: An Almost Depth-First-Search Distributed Graph-Querying System** 209
Vasileios Trigonakis and Jean-Pierre Lozi, *Oracle Labs*; Tomáš Faltín, *Oracle Labs and Charles University*; Nicholas P. Roth, *KUNGFU.AI*; Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Călin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi, *Oracle Labs*
- GLIST: Towards In-Storage Graph Learning**..... 225
Cangyuan Li, Ying Wang, Cheng Liu, and Shengwen Liang, *SKLCA, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China; University of Chinese Academy of Sciences, Beijing, China*; Huawei Li, *SKLCA, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China; University of Chinese Academy of Sciences, Beijing, China*; Peng Cheng Laboratory, *Shenzhen, China*; Xiaowei Li, *SKLCA, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China; University of Chinese Academy of Sciences, Beijing, China*
- DART: A Scalable and Adaptive Edge Stream Processing Engine** 239
Pinchao Liu, *Florida International University*; Dilma Da Silva, *Texas A&M University*; Liting Hu, *Virginia Tech*
- CrystalPerf: Learning to Characterize the Performance of Dataflow Computation through Code Analysis** 253
Huangshi Tian, *HKUST*; Minchen Yu, *HKUST and Huawei Technologies Ltd.*; Wei Wang, *HKUST*
- Controlling Memory Footprint of Stateful Streaming Graph Processing** 269
Pourya Vaziri and Keval Vora, *Simon Fraser University*

Please Don't Chain Me Up: Blockchain and Security

- Avocado: A Secure In-Memory Distributed Storage System** 285
Maurice Bailleu, Dimitra Giantsidi, and Vasilis Gavrielatos, *University of Edinburgh*; Do Le Quoc, *Huawei Research*; Vijay Nagarajan, *University of Edinburgh*; Pramod Bhatotia, *University of Edinburgh and TU Munich*
- Accelerating Encrypted Deduplication via SGX** 303
Yanjing Ren and Jingwei Li, *University of Electronic Science and Technology of China*; Zuoru Yang and Patrick P. C. Lee, *The Chinese University of Hong Kong*; Xiaosong Zhang, *University of Electronic Science and Technology of China*
- ICARUS: Attacking low Earth orbit satellite networks** 317
Giacomo Giuliani, Tommaso Ciussani, Adrian Perrig, and Ankit Singla, *ETH Zurich*
- RainBlock: Faster Transaction Processing in Public Blockchains** 333
Soujanya Ponnappalli, Aashaka Shah, and Souvik Banerjee, *University of Texas at Austin*; Dahlia Malkhi, *Diem Association and Novi Financial*; Amy Tai, *VMware Research*; Vijay Chidambaram, *University of Texas at Austin and VMware Research*; Michael Wei, *VMware Research*
- An Off-The-Chain Execution Environment for Scalable Testing and Profiling of Smart Contracts** 349
Yeonsoo Kim and Seongho Jeong, *Yonsei University*; Kamil Jezek, *The University of Sydney*; Bernd Burgstaller, *Yonsei University*; Bernhard Scholz, *The University of Sydney*

I'm Old But I Learned a New Trick: Machine Learning

- Octo: INT8 Training with Loss-aware Compensation and Backward Quantization for Tiny On-device Learning** . 365
Qihua Zhou and Song Guo, *Hong Kong Polytechnic University*; Zhihao Qu, *Hohai University*; Jingcai Guo, Zhenda Xu, Jiewei Zhang, Tao Guo, and Boyuan Luo, *Hong Kong Polytechnic University*; Jingren Zhou, *Alibaba Group*
- Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism** 381
Saar Eliad, Ido Hakimi, and Alon De Jagger, *Department of Computer Science, Technion - Israel Institute of Technology*; Mark Silberstein, *Department of Computer Science and Department of Electrical Engineering, Technion - Israel Institute of Technology*; Assaf Schuster, *Department of Computer Science, Technion - Israel Institute of Technology*
- INFaaS: Automated Model-less Inference Serving** 397
Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis, *Stanford University*
- Jump-Starting Multivariate Time Series Anomaly Detection for Online Service Systems** 413
Minghua Ma, *Tsinghua University, BNRist*; Shenglin Zhang, *Nankai University*; Junjie Chen, *Tianjin University*; Jim Xu, *Georgia Tech*; Haozhe Li and Yongliang Lin, *Nankai University*; Xiaohui Nie, *Tsinghua University, BNRist*; Bo Zhou and Yong Wang, *CNCERT/CC*; Dan Pei, *Tsinghua University, BNRist*
- Palleon: A Runtime System for Efficient Video Processing toward Dynamic Class Skew** 427
Boyuan Feng, Yuke Wang, Gushu Li, Yuan Xie, and Yufei Ding, *University of California, Santa Barbara*

Thursday, July 15

Can I Come In? It's Raining!: Cloud Computing

- FAASNET: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute** 443
Ao Wang, *George Mason University*; Shuai Chang, *Alibaba Group*; Huangshi Tian, *Hong Kong University of Science and Technology*; Hongqi Wang, Haoran Yang, Huiba Li, and Rui Du, *Alibaba Group*; Yue Cheng, *George Mason University*
- Experiences in Managing the Performance and Reliability of a Large-Scale Genomics Cloud Platform** 459
Michael Hao Tong, Robert L. Grossman, and Haryadi S. Gunawi, *University of Chicago*
- Scaling Large Production Clusters with Partitioned Synchronization** 473
Yihui Feng, *Alibaba Group*; Zhi Liu, Yunjian Zhao, Tatiana Jin, and Yidi Wu, *The Chinese University of Hong Kong*; Yang Zhang, *Alibaba Group*; James Cheng, *The Chinese University of Hong Kong*; Chao Li and Tao Guan, *Alibaba Group*
- Fighting the Fog of War: Automated Incident Detection for Cloud Systems** 489
Liquan Li and Xu Zhang, *Microsoft Research*; Xin Zhao, *University of Chinese Academy of Sciences*; Hongyu Zhang, *The University of Newcastle*; Yu Kang, Pu Zhao, Bo Qiao, and Shilin He, *Microsoft Research*; Pochian Lee, Jeffrey Sun, Feng Gao, and Li Yang, *Microsoft Azure*; Qingwei Lin, *Microsoft Research*; Saravanakumar Rajmohan, *Microsoft 365*; Zhangwei Xu, *Microsoft Azure*; Dongmei Zhang, *Microsoft Research*

SIT, Fido!: Training Machine Learning Algorithms

- Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training** 503
Geoffrey X. Yu, *University of Toronto/Vector Institute*; Yubo Gao, *University of Toronto*; Pavel Golikov and Gennady Pekhimenko, *University of Toronto/Vector Institute*
- Zico: Efficient GPU Memory Sharing for Concurrent DNN Training** 523
Gangmuk Lim, *UNIST*; Jeongseob Ahn, *Ajou University*; Wencong Xiao, *Alibaba Group*; Youngjin Kwon, *KAIST*; Myeongjae Jeon, *UNIST*
- Refurbish Your Training Data: Reusing Partially Augmented Samples for Faster Deep Neural Network Training** ... 537
Gyewon Lee, *Seoul National University and FriendliAI*; Irene Lee, *Georgia Institute of Technology*; Hyeonmin Ha, Kyungeun Lee, and Hwarim Hyun, *Seoul National University*; Ahnjae Shin and Byung-Gon Chun, *Seoul National University and FriendliAI*
- ZeRO-Offload: Democratizing Billion-Scale Model Training** 551
Jie Ren, *UC Merced*; Samyam Rajbhandari, Reza Yazdani Aminabadi, and Olatunji Ruwase, *Microsoft*; Shuangyan Yang, *UC Merced*; Minjia Zhang, *Microsoft*; Dong Li, *UC Merced*; Yuxiong He, *Microsoft*

I Can Smell That Fluffy Was Here: Networks

- Hashing Linearity Enables Relative Path Control in Data Centers** 565
Zhehui Zhang, *University of California, Los Angeles*; Haiyang Zheng, Jiayao Hu, Xiangning Yu, Chenchen Qi, Xuemei Shi, and Guohui Wang, *Alibaba Group*
- Live in the Express Lane** 581
Patrick Jahnke, *TU Darmstadt and SAP*; Vincent Riesop, *SAP*; Pierre-Louis Roman and Pavel Chuprikov, *Università della Svizzera italiana*; Patrick Eugster, *Università della Svizzera italiana, TU Darmstadt, and Purdue University*
- Understanding Precision Time Protocol in Today's Wi-Fi Networks: A Measurement Study** 597
Paizhuo Chen and Zhice Yang, *ShanghaiTech University*
- AUTO: Adaptive Congestion Control Based on Multi-Objective Reinforcement Learning for the Satellite-Ground Integrated Network** 611
Xu Li, Feilong Tang, and Jiacheng Liu, *Shanghai Jiao Tong University*; Laurence T. Yang, *St. Francis Xavier University*; Luoyi Fu and Long Chen, *Shanghai Jiao Tong University*
- Hey, Lumi! Using Natural Language for Intent-Based Network Management** 625
Arthur S. Jacobs, Ricardo J. Pfitscher, and Rafael H. Ribeiro, *Federal University of Rio Grande do Sul (UFRGS)*; Ronaldo A. Ferreira, *UFMS*; Lisandro Z. Granville, *Federal University of Rio Grande do Sul (UFRGS)*; Walter Willinger, *NIKSUN, Inc.*; Sanjay G. Rao, *Purdue University*

I Buried That Bone Here Somewhere: Storage

- Boosting Full-Node Repair in Erasure-Coded Storage.** 641
Shiyao Lin, Guowen Gong, and Zhirong Shen, *Xiamen University*; Patrick P. C. Lee, *The Chinese University of Hong Kong*; Jiwu Shu, *Xiamen University and Tsinghua University*
- KVIMR: Key-Value Store Aware Data Management Middleware for Interlaced Magnetic Recording Based Hard Disk Drive** 657
Yuhong Liang, Tsun-Yu Yang, and Ming-Chang Yang, *The Chinese University of Hong Kong*
- Differentiated Key-Value Storage Management for Balanced I/O Performance** 673
Yongkun Li and Zhen Liu, *University of Science and Technology of China*; Patrick P. C. Lee, *The Chinese University of Hong Kong*; Jiayu Wu and Yinlong Xu, *Anhui Province Key Laboratory of High Performance Computing, University of Science and Technology of China*; Yi Wu, Liu Tang, Qi Liu, and Qiu Cui, *PingCAP*
- ZNS: Avoiding the Block Interface Tax for Flash-based SSDs** 689
Matias Björling, *Western Digital*; Abutalib Aghayev, *The Pennsylvania State University*; Hans Holmberg, Aravind Ramesh, and Damien Le Moal, *Western Digital*; Gregory R. Ganger and George Amvrosiadis, *Carnegie Mellon University*
- MapperX: Adaptive Metadata Maintenance for Fast Crash Recovery of DM-Cache Based Hybrid Storage Devices.** ..705
Lujia Yin, *NUDT*; Li Wang, *Didi Chuxing*; Yiming Zhang, *NiceX Lab, NUDT*; Yuxing Peng, *NUDT*

Friday, July 16

My Tail Never Has Any Latency: OS & Hardware

- Exploring the Design Space of Page Management for Multi-Tiered Memory Systems.** 715
Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn, *Ajou University*
- A Fast and Flexible Hardware-based Virtualization Mechanism for Computational Storage Devices.** 729
Dongup Kwon, Dongryeong Kim, Junehyuk Boo, Wonsik Lee, and Jangwoo Kim, *Seoul National University*
- Fair Scheduling for AVX2 and AVX-512 Workloads** 745
Mathias Gottschlag, Philipp Machauer, Yussuf Khalil, and Frank Bellosa, *Karlsruhe Institute of Technology*
- SKQ: Event Scheduling for Optimizing Tail Latency in a Traditional OS Kernel** 759
Siyao Zhao, Haoyu Gu, and Ali José Mashtizadeh, *University of Waterloo*
- A Linux Kernel Implementation of the Homa Transport Protocol.** 773
John Ousterhout, *Stanford University*

Friends Fur-Ever: Persistent Memory and In-Memory Computing

- Ayudante: A Deep Reinforcement Learning Approach to Assist Persistent Memory Programming** 789
Hanxian Huang, Zixuan Wang, Juno Kim, Steven Swanson, and Jishen Zhao, *University of California, San Diego*
- Tips: Making Volatile Index Structures Persistent with DRAM-NVMM Tiering.** 805
R. Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, and Sumit Kumar Monga, *Virginia Tech*; Hee Won Lee, *Samsung Electronics*; Minsung Jang, *Peraton Labs*; Ajit Mathew and Changwoo Min, *Virginia Tech*
- Improving Performance of Flash Based Key-Value Stores Using Storage Class Memory as a Volatile Memory Extension** 821
Hiwot Tadese Kassa, *University of Michigan*; Jason Akers, Mrinmoy Ghosh, and Zhichao Cao, *Facebook Inc.*; Vaibhav Gogte and Ronald Dreslinski, *University of Michigan*
- First Responder: Persistent Memory Simultaneously as High Performance Buffer Cache and Storage** 839
Hyunsub Song, Shean Kim, J. Hyun Kim, Ethan JH Park, and Sam H. Noh, *UNIST*
- A Case Study of Processing-in-Memory in off-the-Shelf Systems** 855
Joel Nider, Craig Mustard, Andrada Zoltan, John Ramsden, Larry Liu, Jacob Grossbard, and Mohammad Dashti, *University of British Columbia*; Romaric Jodin, Alexandre Ghiti, and Jordi Chauzi, *UPMEM SAS*; Alexandra Fedorova, *University of British Columbia*

Time to File the Claws: Files

XFUSE: An Infrastructure for Running Filesystem Services in User Space 863
Qianbo Huai, Windsor Hsu, Jiwei Lu, Hao Liang, Haobo Xu, and Wei Chen, *Alibaba Group*

MAX: A Multicore-Accelerated File System for Flash Storage 877
Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu, *Department of Computer Science and Technology, Tsinghua University, and Beijing National Research Center for Information Science and Technology (BNRist)*

Z-Journal: Scalable Per-Core Journaling 893
Jongseok Kim and Cassiano Campes, *Sungkyunkwan University*; Joo-Young Hwang, *Samsung Electronics Co., Ltd.*;
Jinkyu Jeong and Euseong Seo, *Sungkyunkwan University*

LODIC: Logical Distributed Counting for Scalable File Access 907
Jeoungahn Park, *KAIST*; Taeho Hwang, *Hanyang University*; Jongmoo Choi, *Dankook University*; Changwoo Min, *Virginia Tech*; Youjip Won, *KAIST*

But You Played with Me Yesterday: Serverless Computing and Consistency

UNISTORE: A fault-tolerant marriage of causal and strong consistency 923
Manuel Bravo, Alexey Gotsman, and Borja de Régil, *IMDEA Software Institute*; Hengfeng Wei, *Nanjing University*

Optimistic Concurrency Control for Real-world Go Programs 939
Zhizhou Zhang, *University of California, Santa Barbara*; Milind Chabbi and Adam Welc, *Uber Technologies*; Timothy Sherwood, *University of California, Santa Barbara*

Faastlane: Accelerating Function-as-a-Service Workflows 957
Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu, *Indian Institute of Science*

SONIC: Application-aware Data Passing for Chained Serverless Applications 973
Ashraf Mahgoub, *Purdue University*; Karthick Shankar, *Carnegie Mellon University*; Subrata Mitra, *Adobe Research*;
Ana Klimovic, *ETH Zurich*; Somali Chaterji and Saurabh Bagchi, *Purdue University*

Message from the USENIX ATC '21 Program Co-Chairs

1. Introduction

Welcome to the 2021 USENIX Annual Technical Conference (ATC).

Running a conference of any size is a significant challenge, one that comes with many opportunities to build upon and perhaps improve what has gone before—and the challenge is all the greater for a conference of the scale and caliber of ATC.

We are fortunate that ATC is sponsored by USENIX, an organization with extensive experience in conference planning. Working with USENIX means joining a well-oiled machine that magically takes care of an almost endless list of tasks that must be completed for a conference to be successful. We are humbled and honored by their trust in asking us to take the helm of USENIX ATC '21 and we are grateful for their invaluable guidance and support throughout the process.

We hope that everyone is as thrilled as we are with the ensuing program. In the rest of this document, we describe the process for USENIX ATC '21, with an emphasis on how we created and managed the program committee.

For USENIX ATC '21, we had to deal with two unusual challenges: first, the ongoing pandemic, and second, the collocation with OSDI '21. The former turned out to be more of a nuisance than a challenge, thanks to the help we received from the USENIX ATC '20 co-chairs, Erez Zadok and Ada Gavrilovska. Ada and Erez had been forced to adapt to the pandemic as it unspooled, switching both their program committee meeting and the conference itself from in-person to virtual. They extensively documented their experience in their own conference message, which we referred to repeatedly, and also offered additional answers to specific questions we had. As a result, we were able to plan a smooth PC meeting (which managed to finish on time!) and are confident that the conference itself will be equally successful.

2. Running a Combined Conference

Navigating the decision to collocate OSDI and ATC was more difficult. In response to requests from the systems research community, USENIX concluded that OSDI needs to be held more often, changing from strict alternation with SOSP to an annual conference. However, for a number of years both SOSP and OSDI have taken place in October; USENIX felt that it would be inappropriate to hold OSDI close to the time when SOSP was scheduled. Instead, after carefully considering the options, the USENIX board decided to experiment with holding OSDI and ATC together.

That presented its own dilemma: since OSDI and ATC are both systems conferences, would they find themselves in competition? How would the the conferences themselves be run? Would either conference experience a significant drop—or a rise—in submissions? Fortunately for us, we had an ace in the hole: the OSDI '21 co-chairs, Angela Demke Brown and Jay Lorch, are two of the nicest and most generous people in systems research. From the beginning, we were able to work closely to ensure a smooth experience.

The first question we had to address, almost immediately after the co-chairs for both conferences were chosen, was submission deadlines. Should the deadlines be the same for both conferences? If not, which conference should “get” to go first? We concluded that OSDI should have an earlier deadline, almost exactly a month before our own. That would allow authors who were unable to make the OSDI deadline to have a significant period of time for completing their papers before they submitted to ATC. It also reduced the burden on the USENIX staff by preventing two simultaneous crunch times.

We then turned to the challenge of selecting program committees for both conferences. OSDI '20 received 400 submissions; USENIX ATC '20 received 350. If those levels continued, we would need a record number of PC members across both conferences. That raised the risk that we would find ourselves competing for PC candidates.

To avoid that possibility, the four co-chairs of the two conferences met several times to discuss PC invitations and create a comprehensive list of candidates. Working from a shared spreadsheet, we created one list for each conference, balancing both lists on a number of axes to ensure that we would have diverse committees that covered a range of viewpoints, areas of expertise, and levels of experience. We gave special attention to including less established researchers, both to advance their careers and to develop new possibilities for future program committees.

Our cooperation in identifying PC members was successful; both conferences were able to assemble strong committees. In the end we wound up with a PC of 120 members: 69 heavy (16-20 reviews expected), 39 light (8-10 reviews), and 12 extended (3-5 reviews).

After considering previous submission levels and growth, we decided not to assume that either conference would experience a drop in submissions; instead we built a program committee that would be able to handle an increase. That proved to be a wise decision.

3. Program Committee

The most important task that we did as USENIX ATC '21 program chairs was selecting and inviting the program committee. The PC had 120 members, with expertise in a wide range of topics. The areas with the highest expertise among the PC members were Distributed Systems, Storage, Cloud, Architecture, OS, Analytics and Machine Learning. Those fields allowed us to cover most of the submissions well, but we could have used more expertise in Quantum Computing. We also were surprised to discover that we had multiple submissions in the area of Low-Earth-Orbit Satellite Networks; we had to seek additional help on that topic.

In addition to covering a wide range of research topics, we also wanted to ensure diverse representation in the PC across several dimensions:

- Junior vs. Senior members of the community. We believe it is important to invite new community members to take part in the review process. Our PC had 34% junior members.
- Academia vs. Industry. USENIX ATC has broad industry impact, so we wanted to have this duality reflected in our PC membership as well. Industry and Industrial Research Labs members constituted 34% of the PC.
- Women and Under-represented Minorities. The PC representation was 22%.
- Geography. 63% of the PC members are located in North America, 24% in Europe, 8% in Asia, and the remaining 5% in other areas.

The PC was split into three categories, depending on the number of reviews the PC members were able to provide, as well as whether they could attend the PC meeting or not: heavy (up to 17 reviews), light (up to 9 reviews), and extended (up to 5 reviews). The heavy PC consisted of 68 members, the light PC 39 members, and the extended PC 12 members.

We found it important for each PC member to provide us information before the reviewing process started. In particular, it simplified the conflict verification process if each PC member updated their own lists of collaborators and other conflicts. Doing this before the submission deadline is ideal, as it provides the authors the opportunity to double-check the identified conflicts through HotCRP. In addition, the PC members rank their topics of interest from the conference topics. While this step is optional for PC members who spent the time to go through the submissions after the deadline and submit their paper preferences, we asked the PC members who did not to make sure they at least had their topics preference filled in. It would have been impossible for us to assign papers to PC members who had expressed neither a paper nor a topic preference. In retrospect, the PC members who actually took the time to bid on (a larger number of) papers also got a better assignment. In addition, it was easier for us to assign papers when extra reviews were needed.

3.1. Submissions Chairs

Following the prior two editions of USENIX ATC, we adopted the practice of having two submissions chairs. Alex Conway (VMware Research) and Chris Stone (Harvey Mudd College) provided invaluable support during the USENIX ATC evaluation period. In particular, the submissions chairs helped with the pre-review checks, verified the authors' reported conflicts, helped with checking the quality of the reviews and took notes during the PC meeting. We would not have been able to put the conference together without their able and tireless assistance.

4. Review Process

Perhaps the most important part of putting a conference program together is the lengthy period of peer review. As is common for most large conferences, we divided the review into several steps.

4.1. Pre-Review Quick Checks

After the deadline passed, we had a total of 341 completed submissions, excluding spurious ones. We divided the papers into four equal groups assigned to the co-chairs and submissions co-chairs for a quick validity check. Each person scanned their pile for several flaws:

- Anonymity violations, both obvious (authors on the title page) and more subtle (self-citation, mention of authors' institutions, etc.).

- Formatting problems not caught by the automated format checker (e.g. unreadable figures).
- Violations of the Call for Papers, such as appendices following the bibliography.
- Other random submission errors, such as accidentally submitting a version of the paper as supplemental material.

During the quick checks, we also identified an oversight in the Call for Papers. A number of authors had included links to online copies of their source code in their papers. Such links had not been prohibited by the CFP, but they created the risk of anonymity violations. In addition, since online resources can be modified at any time, there was no way to ensure that they were representative of work done before the deadline. For those reasons, we decided to ask authors to remove those links.

In the end, we rejected one submission immediately because it was too flawed to be rescued and because it violated the anonymity requirement by including the authors' names. We gave the authors of 67 (!) others a brief window in which to correct the problems; all accepted the offer. Although the CFP clearly stated that format violations would result in instant rejection, we chose to be gentle because the violations appeared to be unintentional mistakes rather than an attempt to evade the rules.

Finally, we also reviewed the papers for undeclared conflicts with PC members. That task was eased because the submission form had included a request for DBLP links for all co-authors. Nevertheless, that task was a major one, and we hugely appreciate Alex and Chris for carrying it out. We added a number of new conflicts into HotCRP, and notified the authors so that they would be more careful in the future.

4.2. Review Timeline

The review process was scheduled over two rounds, with ample discussion time after each round. The discussions were very useful not only because PC members had time to read each other's reviews and reach a common decision, but also because it gave us time to supplement reviews where necessary by asking other PC members or external reviewers. For example, it was necessary to supplement reviews when a paper had lower reviewer expertise, or when the reviews were late or missing. We advise future co-chairs to allow for these buffer periods as well, while still enforcing the round deadlines.

4.3. Review Sequence

Once we had 340 conforming submissions, we assigned reviews to the members of the program committee. In the first round, heavy PC members were assigned 10-12 papers, with proportionally fewer going to light and extended members. Each paper received three first-round reviews. An unexpected issue arose when one member was forced to withdraw from the PC due to serious health issues. We are hugely grateful to Reto Achermann, David Cock, Dave Dice, and Scott Stoller, who were willing to accept an additional assignment on insufficient notice to help out.

After the round-1 reviews arrived, we held brief online discussions to decide which papers would advance to round 2. We took a cautious approach, leaning toward advancement for papers that had received at least one weak accept and papers where no reviewer had rated themselves as having high expertise in the area. At the same time, though, we were cognizant of the fact that we were planning to send round-1 reject notices right away (rather than waiting until after the PC meeting), since rejected authors generally prefer to receive the news early so that they can begin work on a revision. In the end, 178 papers advanced to the second round.¹

Papers rejected in round 1 did not have the option to provide an author response, due to the early notification. We found that a small number of the authors of the papers rejected in round 1 would have preferred to take the option of a later notification, so that they could have the chance to submit a response to the reviews. Thus, future chairs might consider incorporating an option to choose between early notification and response at submission time, or adding a response period to round-1 rejected papers.

For the papers that advanced, we solicited brief (500-word) responses to the reviews from the authors. Unfortunately HotCRP does not enforce the word limit, and some authors went far over. Nevertheless, the program committee found the responses useful in a number of cases, and some papers were accepted at least partly based on the clarifications given in the responses.

Each round-2 paper received at least two additional reviews, so that in the end every paper had at least five². After those reviews were in, we again held online discussions to divide the papers into three categories: clear accepts, clear rejects, and those that needed in-person discussion at the online PC meeting. In several cases we asked for additional reviews from experts in the area, either on the PC or external.

¹ A few were advanced despite low ratings because some reviews were still missing at the deadline.

² The papers that would have been rejected in R1 but for their late reviews were an exception, having only 3 or 4 total reviews.

4.4. PC Meeting

In the end, we wound up with 52 papers that needed discussion by the program committee. We had planned from the first to hold the PC meeting online over a period of two days. Inspired by USENIX’s conference-scheduling practices, we decided to limit the meeting to five-hour slots, running from 7:00 am to noon Pacific Time; those times are acceptable on the West Coast of the U.S., pleasant on the East Coast, and not horrible in Europe. Unfortunately they work less well in some other locales (e.g., 7:30 pm to past midnight in India, midnight to 5:00 am in Korea, and 1:00 am–6:00 am in Eastern Australia).

To deal with the time-zone problem, we polled the PC members to ask where they would be on the days of the meeting. We then hand-scheduled the papers for discussion on a rolling basis, beginning with those that had reviewers with the most challenging time zones so that those people could leave the meeting as soon as possible. The approach worked effectively, but only because of the heroic efforts of our distant PC members to stay up far beyond what most of us would consider a reasonable time.

In past face-to-face PC meetings, a significant amount of time was lost moving conflicted members in and out of the meeting room; HotCRP offers an option to automatically generate a schedule that minimizes such motion. We discarded that option in favor of accommodating time zones; we created a single Zoom breakout room named “Hallway” and used it for conflicts. That worked well; we were usually able to move people into and out of the room in 10–30 seconds so little time was lost.

Given necessary overheads such as introductions and breaks, we wound up with about 10 minutes to discuss each paper. As is common in PC meetings, discussions of the early papers often ran over, but the committee gradually developed a sense of timing so that we finished the first day’s work on schedule. As is also common, the reviewers of a few papers came to an accept/reject decision overnight, and in the end we were able to finish slightly early.

5. Statistics

It has become traditional to offer a number of statistics about the submissions and the review process. Some of the numbers are impressive:

- 341 submitted papers (25 short)
- 64 accepted (1 short), 277 rejected (23.1% acceptance rate)
- 119 PC members wrote 1357 reviews
- 750,218 total words in reviews

As has been common in recent years (and not just in ATC), short papers faced something of an uphill battle, with only a 4% acceptance rate. We encourage future chairs to consider whether the short-paper category is a good idea.

6. Conference Plans

Although we are writing this message well in advance of the conference, we are looking forward to a few changes that we hope will make it run smoothly and improve the virtual conference experience for attendees:

- Before the pandemic, several conferences had run successful experiments with “preview sessions” aimed at new attendees. The goal of these sessions was to provide enough background for the uninitiated to understand a group of papers being presented during the conference. This year, we have asked some PC members to create similar previews that will be posted in advance of the conference itself. The preview session will cover common topics from both USENIX ATC and OSDI.
- One of the most important things that was lost in the shift to virtual conferences was the so-called “hallway track”—the break times during which attendees mingled, met new people, and discussed research ideas. There have been several attempts to find substitutes, including extended Zoom sessions, Zoom breakout rooms, and Slack channels; however, none of these options have been entirely satisfactory. At USENIX ATC and OSDI, we will be experimenting with a new option, ohay.co, which we hope will provide an attractive and fun alternative where small groups can interact with each other in a productive fashion. We want to provide a more interactive environment for attendees to replicate the informal discussions that would normally happen at the conference, as well as provide additional time for authors to answer questions and discuss their work in a smaller group setting.
- Several editions of OSDI have implemented a successful mentoring program—matching students with senior researchers and facilitating 1-1 discussions outside the conference program. A few USENIX ATC ’21 PC members are collaborating with OSDI ’21 PC members to create a common mentoring program for both conferences.

7. Final Thoughts

Co-chairing USENIX ATC '21 has been a challenging, instructive, and rewarding experience. In the end, we are pleased to have been able to create a strong program featuring many new insights and creative ideas. We are grateful to everyone who contributed to creating the program and to all who are still working behind the scenes to ensure that USENIX ATC '21 runs smoothly and provides value to all attendees; to the authors who worked hard on their submissions and final papers; to the program committee and external reviewers, who spent countless hours reading, discussing and selecting the best papers; to the shepherds, who ensured that each paper appears in the program in its best shape; and to the volunteers who are organizing mentoring, networking, and preview sessions jointly with OSDI '21. At last, but most importantly, we are grateful to our submissions chairs, who jumped in at a moment's notice to help out with all the tasks where we felt overwhelmed, and to the wonderful USENIX staff, who directed us and cleared any roadblocks.

We thank everyone for their hard work and dedication. We are humbled by your willingness to go above and beyond to make USENIX ATC '21 a success!

We hope that you enjoy the conference!

USENIX ATC '21 Program Co-Chairs

Irina Calciu, *VMware Research*

Geoff Kuenning, *Harvey Mudd College*

Naos: Serialization-free RDMA networking in Java

Konstantin Taranov¹, Rodrigo Bruno^{2†}, Gustavo Alonso¹, and Torsten Hoeffler¹
¹Department of Computer Science, ETH Zurich ²INESC-ID / Técnico, ULisboa

Abstract

Managed languages such as Java and Scala do not allow developers to directly access heap objects. As a result, to send on-heap data over the network, it has to be explicitly converted to byte streams before sending and converted back to objects after receiving. The technique, also known as object serialization/deserialization, is an expensive procedure limiting the performance of JVM-based distributed systems as it induces additional memory copies and requires data transformation resulting in high CPU and memory bandwidth consumption. This paper presents Naos, a JVM-based technique bypassing heap serialization boundaries that allows objects to be directly sent from a local heap to a remote one with minimal CPU involvement and over RDMA networks. As Naos eliminates the need to copy and transform objects, and enables asynchronous communication, it offers significant speedups compared to state-of-the-art serialization libraries. Naos exposes a simple high level API hiding the complexity of the RDMA protocol that transparently allows JVM-based systems to take advantage of offloaded RDMA networking.

1 Introduction

Managed programming languages, such as Java and Scala, are a common vehicle for developing distributed platforms such as Spark [36], Flink [6], or Zookeeper [11]. However, the high level abstractions available in managed languages often cause significant performance overheads. In particular, to exchange data over the network, Java applications are currently forced to transform structured data via serialization, causing a high CPU overhead and requiring copying the data multiple times. While less of an issue in single-node applications, the overhead is substantial in distributed settings, especially in big data applications. Serialization already accounts for 6% of total CPU cycles at Google datacenters [15].

Data transfer with object serialization/deserialization (OSD) is a complex process involving five steps: **graph**

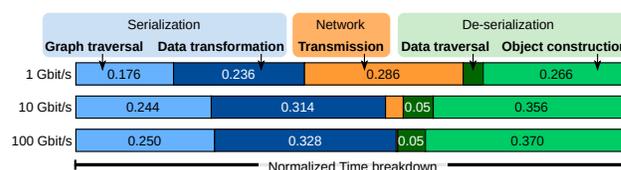


Figure 1: Impact of network bandwidth on OSD time.

traversal to identify all objects that should be serialized; **data transformation** to convert the objects into a byte stream (network-friendly format); **transmission** to send the serialized data over the network; **data traversal** at the receiver to decode the received data; and **object construction** that involves allocating memory and object re-initialization.

To illustrate the CPU overhead caused by OSD, we benchmarked the Kryo [26] serializer and measured its CPU utilization while sending objects over different networks. Figure 1 shows the fraction of time spent on each OSD step for the transfer of an array of 1.28M objects, all of the same exact type. Each object has two fields, each encapsulating a primitive type. Results show that the time spent in OSD increases as networks get faster. For a 10 Gbit/s network, it takes less than 3% of the time to send data over the network, but it takes more than 31%/35% of the time in data transformation/object construction. This discrepancy is even more evident in 100 Gbit/s networks in which the network time drops to less than 0.01% and the time spent on the CPU performing OSD accounts for almost 100% of the transfer time. While networks are getting faster, the pressure is moving away from the network and into the CPU (and memory bandwidth), further aggravating the already well-known CPU-bottleneck problems encountered in distributed applications [22, 32]. Furthermore, existing distributed platforms that heavily rely on OSD are not able to take advantage of faster networks such as RDMA.

With the widespread use of Java in large scale data processing and the increased availability of RDMA, it is time to rethink current OSD techniques so that part of the load of object shifts from the CPU back to the network. In this paper, we aim to develop native runtime support for *serialization-free*

[†]This author was at ETH Zurich during this project.

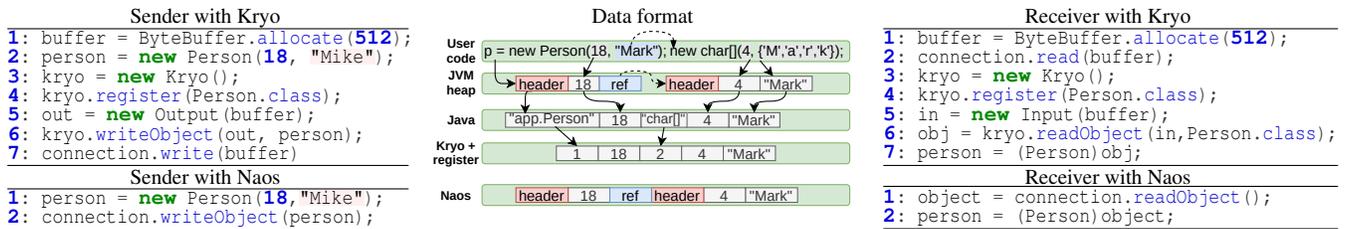


Figure 2: Serialization, Data format, Deserialization for Kryo and Naos

networking that avoids superfluous memory copies and data transformation by sending objects directly from the source heap into the remote heap. Sending and receiving data without data marshalling enables the use of zero-copy RDMA networking, bypassing not only serialization but the need to copy data. Such a design significantly reduces the pressure on the CPU at the cost of higher data volumes to be transferred, since objects are sent in their uncompressed memory format.

To test and evaluate these ideas, we have developed Naos (Naos stands for Not Another Object Serializer), a library and runtime plugin for OpenJDK 11 HotSpot JVM that allows objects in the source heap to be directly written into a remote heap, avoiding data transformations and excessive data copies. Naos is designed to accelerate object transfers in distributed applications by taking advantage of RDMA communication (although it also supports conventional TCP sockets).

Naos allows applications to directly send objects without employing serialization libraries. Its API requires no type registration nor serialization snippets, guaranteeing developers a close to zero effort when building systems using Naos. Finally, Naos is the first (to the best of our knowledge) library integrating RDMA into JVM allowing the user to communicate on-heap objects transparently, thereby easing the adoption of RDMA networking by JVM-based distributed applications. Our evaluation shows that Naos provides a 2x throughput speedup over serialization approaches for transferring contiguous objects and for moderately sparse object graphs. Naos improves latency-sensitive applications such as RPCs by providing a 2.2x reduction in latency.

Contributions. Naos is the first *serialization-free* communication library for JVM that allows applications to send objects directly through RDMA or TCP connections. Naos unlocks efficient asynchronous RDMA networking to JVM users hiding all the burden of low-level RDMA programming from the users, thereby facilitating the adoption of RDMA. For that, Naos solves several complex design issues such as sending unmodified memory segments across Java heaps without employing intermediate buffers, and interacting with concurrent garbage collection without compromising JVM’s memory safety. For the first issue, Naos proposes a novel algorithm that writes objects directly to the remote heap and makes them valid on the receiver’s address space (§3.3). For the second one, Naos proposes techniques preventing a concurrent JVM garbage collector from moving unsent objects

that may be accessed by RNIC and from accessing unrecovered received objects (§3.2). Finally, Naos enables pipelining communication and serialization, which was previously impossible with the OSD approach (§3.4).

2 Object Serialization

Overview. Many third-party libraries [12, 16, 26] have been developed to perform OSD in Java. Some of them provide Java bindings for popular cross-language OSD approaches (e.g., Protobuf [12]), allowing serializing arbitrary data structures into well-defined messages that can then be exchanged using any network protocol. While remaining independent of programming languages or operating systems, such libraries suffer from low performance [21]. Therefore, JVM-based big-data applications (e.g., Spark, Flink) rely on specialized libraries such as Kryo [26], designed specifically for JVMs.

Figure 2 presents a serialization example of a Java object and its data formats: memory layout (JVM heap), and serialization formats (Java, Kryo). All Java objects start with a JVM-specific header (red) followed by a number of primitive (gray) or reference fields (blue). The object of type `Person` has one primitive `int` field followed by a reference field to a character array (`char[]`). The character array starts with the length of the array followed by all characters.

Serializing an object involves traversing the object graph starting from that object and, upon visiting each reachable object, copying all primitive fields into the pre-allocated byte buffer. During native Java serialization, headers are replaced by class descriptors in textual format (`app.Person`) and field references are replaced by the contents of the pointed object. Deserialization follows a similar logic; upon visiting a serialized object, a new object must be allocated, and all primitive fields are copied out of the buffer into the allocated object.

Kryo. Kryo [26], one of the most widely used OSD libraries, addresses some limitations of native Java serialization by requiring manual registration of classes to achieve a more compact representation of the serialized data. Figure 2 shows a serialized data format in Kryo with class registration (Kryo+register). Kryo can represent all primitive types and classes using integer identifiers, thereby reducing the amount of space needed for storing type names. Although the class registration is trivial in this example, this task is cumbersome

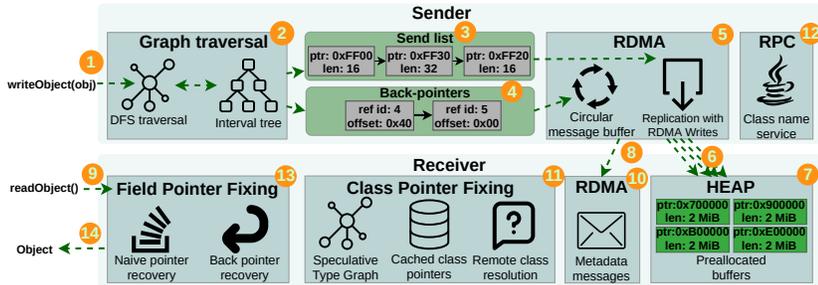


Figure 3: Naos' workflow for sending and receiving a Java object.

for applications with hundreds of data types. Compared to Kryo, Naos provides a cleaner interface (see Figure 2) with no need for developer involvement. To send a Java object, one can directly write it (`writeObject`) to the network. The receiver can directly read the object with `readObject`.

Accelerated OSD. To address the overhead of having to transform the data, Cereal [13] and Optimus Prime [24] resort to dedicated hardware accelerators for OSD. These accelerators are co-designed with the serialization format to parallelize the OSD process. Even though their data formats are not portable across different JVMs, their simulation results promise 15x speedup in serialization throughput on average over Kryo at the expense of requiring specialized hardware.

Zero-transformation OSD. The trade-off portability vs. performance is also exploited by the serialization library Skyway [21]. By dropping portability, Skyway manages to partially avoid data transformations and object construction by serializing Java objects in their JVM formats, i.e., the objects are written to communication buffers in the same binary format they are stored in the heap. Like Skyway, Naos sends objects in the JVM heap format, assuming that communicating parties run on the same JVM software. Unlike Naos, however, Skyway is a *serialization library* requiring to copy objects to and from communication buffers. Naos, on the other hand, completely removes the need to explicitly serialize and deserialize objects to send objects between Java heaps even with RDMA. What is more, Skyway's memory management prevents the use of RDMA networking (§4).

Naos integration and applicability. *Naos is not a serialization library.* Naos only covers end-to-end transfers (see Table 1) and cannot replace OSD in systems that do not use it for communication (e.g., for writing objects to disks). Naos has been primarily designed for future systems that want to take advantage of serialization-free zero-copy RDMA networking.

In several existing Java frameworks the main obstacle to using Naos is that some of these systems do not consider the possibility to send objects without serialization. For example, Spark and Hadoop completely decouple serialization from communication: their serialization modules are designed to serialize objects only to files, and their shuffle modules are designed to communicate only files. Such file-centric design

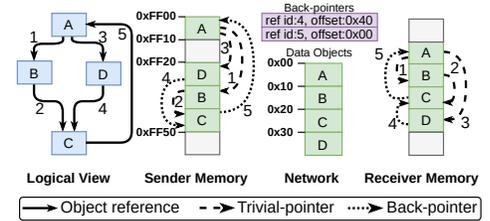


Figure 4: Object views of Naos' graph traversal and pointer recovery.

simplifies inter-node communication, as processes can share file descriptors instead of sending data, and helps to reduce memory usage by dumping data to disks. However, it makes integrating Naos very difficult. For such use-cases, conventional OSD libraries are a better fit than Naos if a redesign for true zero-copy is infeasible.

Table 1: APIs of Naos RDMA.

API	Description
<code>void writeObject(Object)</code>	Blocking send of a single object
<code>Object readObject()</code>	Blocking read of an object from heap
<code>boolean isReadable()</code>	Check whether an object can be read
<code>long writeObjectAsync(Object)</code>	Nonblocking send of a single object
<code>int waitHandle(long)</code>	Wait for a send request to complete
<code>int testHandle(long)</code>	Tests completion of a send request

3 System Overview

Naos allows Java applications to send/receive objects directly through RDMA or TCP connections. Naos uses a collection of algorithms and data structures to efficiently transmit large complex data structures. Figure 3 presents a graphical overview of Naos' workflow, including the main algorithms and data structures. An object transfer starts with a `writeObject` (1) triggering a DFS graph traversal (2) (§3.1). During the traversal, pointers to already visited objects are detected using an interval tree. After the traversal, both the objects (3) and metadata (4) are sent over the network using RDMA (5) (§3.2). Naos uses a circular message buffer to send metadata (8) and writes objects directly to the remote heap (6). Upon reception of the data and metadata, the receiver starts recovering (§3.3) the object graph by fixing class pointers (11) and field pointers (13). Once pointers are fixed, the head of the object graph is returned (14) to the caller of `readObject` (9).

The `writeObject` call in Naos is blocking, that is, the call returns once the object transmission is completed. It ensures that the object is received by the destination. In contrast to the classical TCP/IP semantics, all RDMA operations are executed asynchronously by design, allowing overlapping computation with communication. Naos also provides a nonblocking `writeObjectAsync` call enabling asynchronous communication for RDMA connections (§3.2). The nonblocking call initiates the send operation but does not fully complete it.

Instead, it returns a request handle, that is used by a user to wait for the completion using `waitHandle` call or to verify whether the request is completed using `testHandle` call.

Structure	The length of the send list				Traversal time (us)			
	(1-0-0)	(1-1-0)	(1-2-0)	(1-1-1)	(1-0-0)	(1-1-0)	(1-2-0)	(1-1-1)
BFS	1	2048	3072	3072	42	194	315	271
DFS	1	2	2	1	42	57	74	76

Table 2: Graph traversal of the object *array*.

3.1 Object Graph Traversal

Java objects can contain reference fields pointing to other Java objects and therefore, when an object is passed as an argument to `writeObject` ①, all objects reachable from it need to be sent. To find all objects reachable from a particular object, Naos traverses the object graph ② in Depth-First-Search (DFS) order. Figure 4 illustrates a simple example of an object graph’s (*Logical View*), sender memory layout, format sent over the network, and receiver memory layout. The *sender memory* starts at address `0xFF00` and all objects occupy 16 bytes. Edges are numbered according to DFS order.

When an object is visited for the first time, it is included in the *Send list* ③: a list of memory blocks that will be sent over the network. Each memory block has two elements: the starting virtual address, and the length. The send list contains objects ordered according to DFS order, and the objects are sent in this order over the network. Naos also merges the memory blocks that are adjacent in the send list to reduce its length. For that, during traversal Naos checks whether a new visited memory block is a continuation of the last block of the send list: if yes, then Naos increases the length of the last block, otherwise, Naos adds a new block to the list. The resulting send list is presented in ⑤, which contains three elements: for object *A*, for objects *B* and *C* as they are adjacent in memory and in DFS order, and for object *D*.

DFS vs BFS traversal. Even though Skyway [21] uses BFS traversal for serialization, Naos exploits DFS due to the fact that Java objects are constructed in DFS order (i.e., a JVM first allocates memory for an object and then recursively for all its fields). Thus, DFS traversal has better memory locality that can be illustrated by traversing an object *array* from the following code snippet. Let us consider a class *Person* that has different graph structures denoted as (L0-L1-L2), where *Li* is the number of objects on the level *i* of the object graph (e.g., the object in Figure 4 has structure (1-2-1)).

```

1: Person[] array = new Person[1024];
2: for(int i=0; i<1024; i++)
3:   array[i] = new Person();

```

Table 2 reports the length of the send list after BFS and DFS traversals and corresponding traversal time for several object graphs. The data shows that for complex graph structures DFS provides much shorter send lists and faster traversal time.

Back-pointers. Naos sends objects directly from one heap to another. As a result, objects are sent containing pointers

Algorithm 1 Was object *o* already visited?

```

1: if o.addr = curr.addr + curr.len ∧ o.addr ≠ next.addr then
2:   curr.len ← curr.len + o.size ▷ hot-path
3:   return false
4: if o.addr > curr.addr ∧ o.addr < next.addr then
5:   curr ← tree.insert_before(next, o) ▷ warm-path
6:   return false
7: node, success ← tree.insert(o) ▷ cold-path
8: if success then
9:   curr ← node
10:  next ← curr.next()
11:  return false
12: return true ▷ is a back-pointer

```

that are valid only in the sender address space, but not in the receiver’s. Naos addresses this problem by sending extra metadata along with data objects, which is used by the receiver to efficiently recover the pointers (§3.3).

Naos is designed to send as little metadata as possible. The metadata contains a 24-byte header with object and metadata sizes and, if present, pointers to already visited objects ④. These pointers are redundant edges after building a spanning tree over the object graph using DFS. We call them *back-pointers* since they always point to already visited objects in the send list (see Figure 4). For each back-pointer, a reference identifier representing the order by which the reference was visited in DFS order, and an offset within the send list where this reference should point to are sent to the receiver as metadata. In our example in Figure 4, only references 4 (*D* → *C*) and 5 (*C* → *A*) are sent.

All edges of the spanning tree (we call them *trivial-pointers* for simplicity) can be automatically inferred during a DFS traversal in the receiver (§3.3). This allows Naos to send no information about *trivial-pointers* resulting in a massive reduction of metadata sent over the network. Note the graphs without cycles do not contain back-pointers, which covers the vast majority of the most popular Java data structures.

Back-pointer/Cycle detection. To detect pointers to already visited objects (i.e., back-pointers), Naos uses a *memory interval tree* that keeps tracks of all visited memory intervals during DFS traversal. The interval tree is implemented using a red-black tree, which is selected over a hashtable (as Java and Kryo do) for two reasons. First, for large data structures, the hashtable grows (one entry per visited object) to large sizes and will lead to expensive lookups due to hash collision. Second, references to already visited objects are very rare and references pointing to objects in nearby memory positions are common in most Java popular data structures. Therefore, an interval tree, in most cases, contains a few large memory intervals, thereby ensuring fast lookups. We further optimize our interval tree by providing different fast paths.

Algorithm 1 presents how Naos decides whether a particular object *o* has been already visited. Two helper variables are used: *curr* points to the last node inserted into the tree; *next* points to the tree node that follows *curr* in the tree. All

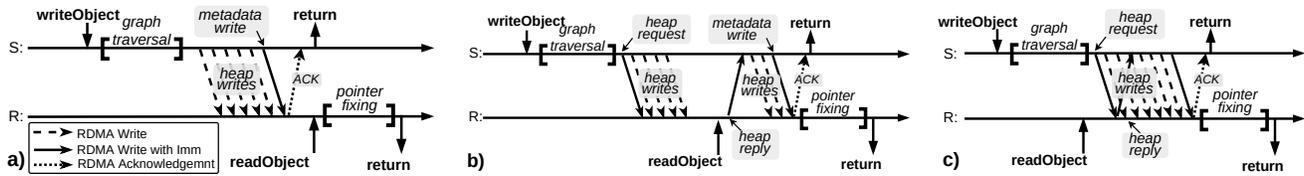


Figure 5: Blocking communication mechanism for three scenarios: (a) the sender can fit data to the pre-allocated receiver heap; (b,c) the sender needs to request extra heap memory. The receiver was not ready to receive data in (b), and it was ready in (c).

tree nodes keep an initial address `addr` and its length `length`. If the object's address is adjacent to the last memory interval inserted into the tree, the insertion is performed in $O(1)$ time (*hot-path*). If the memory pointer is higher than the current tree node and lower than the *next* tree node, then insertion is performed in $O(1)$ (*warm-path*), unless the tree needs to be re-balanced, taking $O(\log(n))$ time. Otherwise, the memory pointer is inserted in the tree in $O(\log(n))$ time (*cold-path*).

As a comparison, Skyway does not use complex structures for cycle detection and simply extends the JVM header of Java objects by 8 bytes. Even though it ensures that the newness of an object can be checked in $O(1)$, it results in a 15.4% increase in memory usage [21].

3.2 Network exchange of on-Heap Objects

Naos adds native RDMA communication to JVM without compromising JVM's memory safety. Naos' interface does not expose explicit RDMA access to the remote or local heap memory. Instead, its API allows only sending and receiving Java objects, hiding all the burden of low-level RDMA programming from the user. Internally, though, Naos fully relies on efficient one-sided RDMA communication to completely avoid redundant data copies. Naos also supports TCP for sending objects directly from its heap, but the use of RDMA requires overcoming peculiarities of managed languages such as concurrent garbage collection.

Blocking RDMA protocol. This section describes the *blocking* RDMA protocol for a single connection. All connections are handled independently and do not share resources. The core idea of Naos RDMA is that the receiver pre-registers buffers of fixed size in its heap and registers them for RDMA Write access. The sender uses RDMA Writes 6 to write the objects from its local heap directly to the known reserved buffers in the remote heap 7. The metadata is sent separately using a circular buffer 8 for RDMA messaging [8, 23].

The protocol allows the sender to start writing memory to the remote heap even if the receiver did not call `readObject`, as illustrated in Figure 5(a). The sender can continue writing the data while it has enough free remote memory. Once the sender completes writing all objects to the remote heap using RDMA, it sends a separate completion message with metadata via the circular message buffer 8. The remote circular buffer is filled using *RDMA Write with immediate data*, which

generates a completion event on the receiver after the write completes. The sender can unblock from sending once it receives an acknowledgment from the network indicating that all data has been written to the receiver. The acknowledgment is generated by the network and does not require the receiver's interaction. The receiver fetches the received object when it calls `readObject`, after all pointers are recovered (§3.3).

Sender's heap management. Naos utilizes *object pinning* to prevent a JVM garbage collector (GC) from moving objects until they are fully transmitted by the RNIC. Object pinning is already offered by some garbage collectors, such as Shenandoah [9]. Shenandoah is a high-performance GC that is supported by upstream OpenJDK. Besides object pinning, Naos also utilizes Shenandoah's memory allocator, that maintains the heap as the collection of fixed size Shenandoah regions. To pin and unpin objects efficiently, Naos pins whole Shenandoah regions containing the affected objects instead of pinning individual objects. During a send request, Naos pins and remembers all affected Shenandoah memory regions. Once the request completes, Naos unpins the regions associated with the request. Shenandoah allows pinning a region multiple times, and each region needs to be unpinned as many times as it has been pinned, thereby successfully preventing Naos from accidentally unlocking the GC for unsent objects.

RNICs cannot simply send data from any buffer and communication buffers must be registered at the RNIC*. Thus, the sender must register the memory addresses of all objects it needs to send. However, RDMA memory registration is an expensive process that may take hundreds of microseconds for a single buffer [14, 20, 30]. Therefore, naive registration of all objects from the send list may completely cancel all performance advantages of RDMA. Naos addresses this issue by registering large fixed-size memory regions (i.e., Shenandoah regions) where the objects are allocated. It enables reusing a single memory registration for all objects stored in it, exploiting spatial locality. Naos also caches memory registrations to reuse them later for future sends, exploiting temporal locality.

Receiver's heap management. When the sender runs out of the remote buffers for writing, it sends a request to the receiver to register more on-heap memory, as illustrated in Figure 5(b,c). Thus, the sender can block until the receiver

*Modern RNICs support implicit on-demand paging (ODP) [17] that removes the need to register buffers. In our preliminary experiments, however, ODP performed worse than conventional explicit memory registration.

replies with new heap buffers, as in Figure 5(b). However, when the receiver is ready to receive data it can immediately reply to the heap request and do not obstruct the sender as in Figure 5(c). The receiver can reply to heap requests when it calls `readObject` or `isReadable`. During these calls, Naos checks for received requests by polling completion events from the RNIC. The process of handling requests is invisible to the caller, which hides the complexity of the underlying protocol from the user.

Upon receiving a heap request, the receiver allocates a new Java byte array buffer of fixed size inside the Java heap and *registers its payload* for RDMA Write access and replies with the RDMA address of the registered buffer. To prevent the GC from moving the reserved on-heap buffers, Naos utilizes *object pinning* offered by Shenandoah [9]. Importantly, the sender writes data to the *payload* of the pre-allocated byte array as it prevents the GC from reading invalid data. The main reason for that is that the unrecovered received objects have invalid class and object pointers (§3.3). Thanks to this enclosure, the GC observes only the array and skips reading objects stored in the payload.

The sender fills the remote buffers in the order it received them from the receiver, constituting a queue of remote heap buffers. Since pre-registered RDMA heap buffers are of fixed size, the sender is not always capable of fully utilizing them. To address this issue, the sender informs the receiver about how many bytes were unused in each *finalized* heap buffer by sending *heap truncate request*. A buffer becomes finalized when the sender jumps to the next buffer in the queue. After receiving the data, the receiver revokes RDMA access to finalized buffers and then unpins them to enable the GC for received objects. It also deallocates unused memory of the finalized buffer and removes the array header to make all received objects visible to the GC.

Nonblocking object sending. The main difference between the blocking `writeObject` and the nonblocking `writeObjectAsync` is that the latter returns right after the dispatching *metadata write* request to the device. The nonblocking call submits all communication requests to the RNIC but does not wait for a network acknowledgment. Instead, Naos returns a request handle that can be used by an application to confirm the delivery of the object using `testHandle` call. Compared to the blocking call, Naos prevents the GC from moving affected objects even after the call returns. Naos *pins* the affected objects before exiting the JVM, and unpins them later once the corresponding acknowledgment is received.

Naos TCP. Naos supports sending objects directly from the heap using TCP as well. Unlike RDMA connections, a traditional TCP socket connection has a single datapath. Thus, to send the objects to the remote heap, the TCP sender first writes the metadata to the socket and then all elements of the send list. The receiver first reads metadata to a temporal buffer from its socket, then, to avoid redundant data copies, it directly reads the data from the socket to the heap. For

that, it allocates a byte array buffer of the required size inside the Java heap, and then reads the data from the socket to the payload of the allocated buffer.

Network buffering. Naos is designed to send data directly from the heap without intermediate buffering. However, the size of a JVM object can be as small as 24 bytes. Thus, a highly sparse object graph can result in a lot of small writes to the network, which can significantly reduce the network performance. To address this issue, Naos may buffer small objects before sending them to the remote heap. Large objects are still sent directly from the heap. Naos sends buffered objects once it batches enough bytes to utilize the network, or when a large object needs to be flushed to preserve DFS object order (§3.1).

An alternative approach is to use scatter-gather capability of RNICs [18] for RDMA networking and scatter-gather I/O for TCP sockets. The scatter-gather networking enables building a network message from multiple buffers without intermediate buffering. The current version of Naos does not implement it, but it is an interesting direction for future research.

Memory safety of Naos. Naos uses reliable transport to ensure the delivery of transmitted data. Naos materializes only fully received objects, which prevents returning partially received objects from a faulty sender. Faulty sends can be detected during graph recovery from the network errors provided by the reliable transport. If an error is detected, the receiver revokes RDMA access to pre-allocated buffers and deallocates the unused memory.

Naos' implementation follows all security advice related to RDMA networking [25, 31], therefore, we believe that Naos does not open security breaches. In particular, the pre-allocated heap buffers are not shared between connections preventing remote JVMs to access buffers of each other. In addition, each sender registers its heap only for local read access preventing other remote JVMs to access it. Finally, remote read access is always disabled, and Naos only temporarily enables write access to pre-allocated in-heap buffers, which are private for each sender. Once the in-heap buffer is full, the write access is revoked.

For compatibility between communicating applications, Naos requires that communicating JVMs have the same memory layout of in-heap objects. This can be achieved by running the same JVM with the same settings including GC.

3.3 Object Graph Recovery

Naos sends unmodified memory segments from one heap to another. As a result, objects are sent containing pointers that are valid only on the sender address space, but not on the receiver's. Naos' graph recovery algorithm overwrites these pointers making them valid on the receiver's address space. Java objects have two types of pointers: **class pointers** and **object pointers**. Class pointers point to JVM-internal data structures that describe Java types. Object pointers are

Algorithm 2 Object Graph Recovery

```
1: buffer                                ▷ the buffer with received objects
2: refid ← 0                              ▷ the number of traversed references
3: offset ← 0                             ▷ current offset in the receive buffer
4: stack.push(new_field(), new_hint())    ▷ push dummy field and hint
5: while stack.is_not_empty() do
6:   field, hint ← stack.pop()
7:   FIX_FIELD_POINTER(field, hint)
8:   refid ← refid + 1
```

Phase 1 – Fix Field Reference

```
9: procedure FIX_FIELD_POINTER(field, hint)
10: if refid = cur_back_pointer.id then                                ▷ a back-pointer
11:   field.ptr ← buffer + cur_back_pointer.offset
12:   cur_back_pointer ← get_next_back_pointer()
13: else                                                                    ▷ a trivial-pointer
14:   obj ← (obj)(buffer + offset)
15:   field.ptr ← obj
16:   FIX_CLASS_POINTER(obj, hint)
17:   ITERATE_FIELDS(obj, hint)
18:   offset ← offset + obj.size
```

Phase 2 – Fix Class

```
19: procedure FIX_CLASS_POINTER(obj, hint)
20: if hint.rem_class = obj.class then
21:   // hint is correct, do nothing                                       ▷ hot-path
22: else
23:   if class_cache.contains(obj.class) then
24:     new_hint ← class_cache.get(obj.class)                                ▷ warm-path
25:     hint.update(new_hint)
26:   else
27:     new_hint ← class_service(obj.class)                                ▷ cold-path
28:     class_cache.put(obj.class, new_hint)
29:     hint.update(new_hint)
30:   obj.class ← hint.loc_class
```

Phase 3 – Iterate Fields

```
31: procedure ITERATE_FIELDS(obj, hint)
32:   for field, field_hint in hint.fields do
33:     stack.push({obj + field.offset, field_hint})
```

reference fields that point to other on-heap Java objects.

Naos uses a recovery approach different from the one used in Skyway [21]. Since Skyway copies objects to communication buffers, it can afford modifying data before sending. Thus, Skyway simply replaces class pointers with integers (as Kryo does) and object pointers with their relative offsets within the communication buffer. Such design allows the receiver to simply replace class integers with corresponding class pointers and relative object offsets with corresponding absolute addresses. Unlike Skyway, Naos sends objects directly from the heap using RDMA requiring more sophisticated algorithm for pointer fixing in return for not requiring data copying.

Algorithm 2 describes the Naos’ graph recovery approach that starts with a DFS traversal of the object fields (lines 5-8). The traversal is initialized by pushing a *dummy* field pointing to the first received object. The graph recovery terminates when the DFS *stack* is empty. At that point, all pointers are valid in the receiver’s heap and the first object can be safely returned to the user.

Fixing Field References. For every object field, the algo-

rithm applies *FIX_FIELD_POINTER* procedure, which investigates whether the tested reference is a *back-pointer* or a *trivial-pointer* by checking whether the received metadata contains the current reference ID (line 10). For *back-pointers*, the *offset* associated with the current pointer is used to fix the reference. If the reference is a *trivial-pointer*, the new memory address can be determined by just using the current *offset* in the *receive* buffer (line 14). For a trivial-pointer, the next step is to fix the class field of the pointed unvisited object (line 16). Note that Naos sends no metadata for trivial pointers, since the sender and the receiver traverse the graph in the same DFS order, providing a significant reduction in metadata size.

Fixing Class References. Updating class pointers is a particularly expensive operation if not designed carefully, since the class pointer needs to be fixed for every object. To achieve high performance, Naos proposes a 3-way approach:

Class Service (cold-path) is an RPC service [12] that is started upon creation of a Naos connection. Once a receiver needs to determine the class of a particular sender’s class pointer, it issues an RPC request to the sender to translate the pointer to the full class name. The full class name can be used locally to query local JVM internal data structures.

Class Map (warm-path) is a per-connection table that caches all class translations. However, accessing a table for every object reference still produces a large overhead, especially in large graphs. To overcome this limitation, Naos proposes the use of Speculative Type Graphs (STG), a type of polymorphic cache inspired by [10].

STG (hot-path) is a data structure that dynamically captures type relations in the object graph, providing a translation *hint* for each class pointer. Each STG hint caches: i) a translation between a local and remote class pointer; ii) class description including object fields; iii) pointers to other hints for each field allowing to build hints recursively (lines 32-33). Using STG, Naos can speculate on the type of a particular object using a hint. If the hint is correct the class translation and retrieval of a class descriptor takes $O(1)$ time (line 20). Speculation might fail due to type polymorphism in Java (line 22) and, in that case, the cache is used for resolving the class pointer and the STG is updated (line 25) with the new translation hint. In practice, however, most data structures have very regular type graphs allowing the STG to guess correctly most of the times.

After the class pointer of an object is fixed, Naos iterates all its reference fields (line 17). Naos utilizes object’s class pointer translation hint to create translation hints for its reference fields (lines 32-33), which are then pushed into the *stack* together with the corresponding object reference.

3.4 Overlapping network and graph traversal

An important disadvantage of conventional serialization approach is that it does not support overlapping serialization and communication: an object must be fully serialized before sending it over a network. Similarly, the receiver cannot start

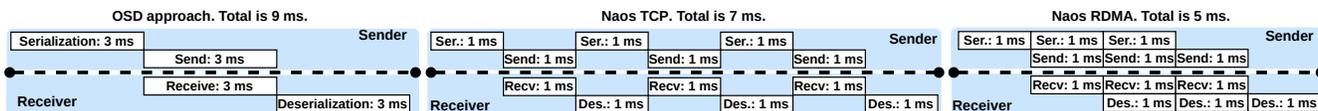


Figure 6: The communication benefits of Naos’ pipelining compared to the conventional OSD approach.

deserialization unless it receives all the data (see Figure 6). As a result, applications can suffer from high end-to-end latency for large object graphs.

Naos supports pipelining graph traversal with communication on the sender and pointer fixing with object receiving on the receiver. Both Naos TCP and Naos RDMA benefit from pipelining as it allows the receiver to start pointer fixing of *partially received object graphs*, thereby reducing end-to-end latency. Using offloaded RDMA communication, Naos RDMA can continue traversing the graph after submitting write requests to the RNIC, thereby overlapping communication and graph traversal on both the sender and the receiver.

Pipelining in Naos is implemented by pausing the object traversal and sending partial graphs to the remote heap. A partial graph contains only objects and back-pointers found at a given traversal stage. The receiver can read the partial graph and start pointer fixing. Once all received objects are traversed, the receiver reads the next fragment of the graph.

Figure 6 illustrates how Naos with pipelining improves communication latency of large object graphs compared to the OSD approach. The OSD approach cannot break serialization of a single graph, which results in 9 ms latency. Naos TCP can send partially traversed graphs reducing the latency by 2 ms, but cannot overlap computation with communication. Naos RDMA enables overlapping communication and graph traversal, which reduces the latency by another 2 ms.

4 Evaluation

We evaluate the performance of Naos[†] and compare it with Java, Kryo, and Skyway[‡] serialization engines using four different classes of workloads. First, the performance of Naos is studied by transferring data structures that are commonly used in distributed applications. The goal is to measure the performance benefits of the different techniques proposed in Naos and the trade-offs involved depending on the shape of object graph. In addition, it also shows the impact of using RDMA instead of TCP. Second, we study the role of data streaming and pipelining in OSD performance. Then, we show results for integrating the Naos library into Apache Dubbo [2], a high-performance RPC framework developed in Java, to show the impact of Naos on RPC workloads. Lastly,

[†]The source code is available at <https://github.com/spcl/naos/>.

[‡]We could not compare with the original Skyway as it is not open-source. Therefore, we re-implemented Skyway following the instruction provided in the paper [21]. Note that we did not extend object headers by 8 bytes for cycle detection and simply evaluated Skyway without cycle detection.

we use a map-reduce implementation of PageRank to measure the performance of Naos for data processing workloads.

Experimental setup. All experiments were performed on a cluster of 4 nodes interconnected by 100 Gbit/s Mellanox ConnectX-5 NICs. Each node is equipped with an Intel(R) Xeon(R) CPU 6154 @ 3.00 GHz and 384 GB of RAM.

Implementation details. Naos is implemented and tested for OpenJDK HotSpot 11.0.6 [1], a widely-used production JVM. Naos does not require changes to the internals of the JVM and is implemented as a JNI plugin and a Java-level library that allows users to write objects directly to TCP and RDMA connections. Naos TCP provides constructors to create a Naos connection from TCP connections of various network libraries (e.g., `java.net.Socket`). Naos RDMA does not rely on existing JVM RDMA libraries and fully implements a specialized RDMA network library including an API to create and connect RDMA endpoints. Our plugin is implemented in Java and C++ and depends on: *libibverbs*, an implementation of the RDMA verbs, and *librdmacm*, an implementation of the RDMA connection manager.

RDMA communicators for Java and Kryo serializers have been implemented using Disni [27] RDMA library, a high-performance Java RDMA library that encapsulates native C RDMA verbs API. The Disni library is used by Java applications such as Spark [19], Crail [29], and DaRPC [28]. Note that Skyway cannot be used with existing RDMA libraries, including Disni, as these libraries can only work with specialized off-heap memory residing outside of the Java heap memory, whereas Skyway requires the memory buffers reside inside the heap memory to deserialize objects. These limitation stems from the fact that garbage collection can move on-heap buffers while they are being accessed by the RNIC.

In all experiments, the JVM was configured with default parameters and enabled Shenandoah garbage collector as it is the only collector that is currently supported by Naos. Shenandoah was configured with 32 MiB memory regions. Naos was configured with 20 MiB receive buffers. If not stated differently, Naos and all serialization algorithms were deployed without graph cycle detection and with no pipelining (§3.4).

4.1 Serializing Java Data Structures

The performance of OSD approaches is measured using three data structures that are among the most common serialized data structures in real-world workloads deployed in platforms such as Spark, Hadoop, and Flink: a) an array of `float` primitive types, which is common for machine learning workloads;

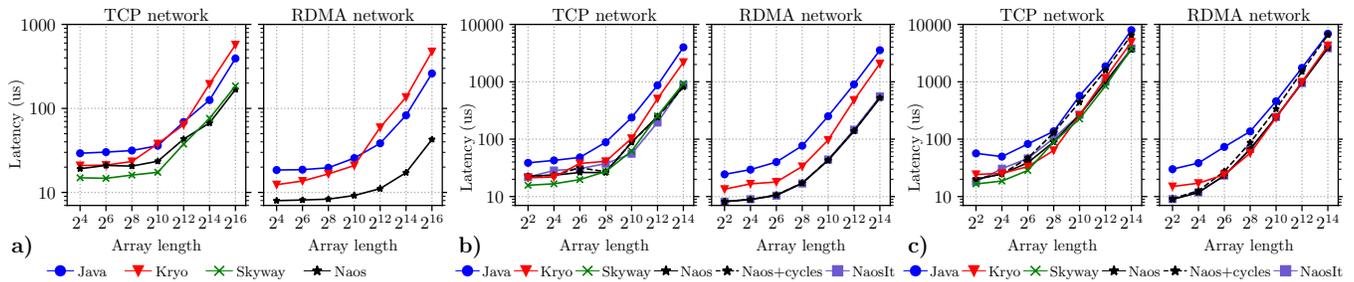


Figure 7: Latency in us for **a)** an array of floats **b)** an array of Points **c)** an array of Pairs. The y-axis is in log scale.

b) an array of `class Point` containing only two primitive types, which represents a 2D Euclidean point; **c)** an array of `class Pair` containing an integer and a char array, which represents a key-value pair, in many algorithms such as Word Count. In our experiments the char array had length 5, the average word length in the English language.

Benchmarks are carefully designed to guarantee the optimal configuration of all serializers. In particular, for Java, Kryo, and Skyway, all buffers are pre-allocated with the correct size to avoid re-allocation and memory copies during the serialization process. Besides, all types were pre-registered in Kryo to guarantee maximum data format compression. Measurements are taken after a JVM warmup (of at least 100 ms) until convergence of the JIT compiler to achieve maximum performance. All experiments run in complete isolation for several seconds and the aggregated statistics are reported.

Latency. Figure 7 shows the average latency of transferring the aforementioned data types with increasing their size.

Naos performs excellently for contiguous data structures such as the array of float, as it can send them from the heap without making extra copies and using fewer RDMA requests. For comparison, Kryo, Java, and Skyway must first serialize objects to a dedicated send buffer. RDMA-Naos' latency can be as small as 8 us, which is at least a 2x and a 2.4x improvements over Kryo and Java serializers, respectively, for small arrays, and at least a 4.5x for large arrays. For example, Naos RDMA needs only 42 us to send 2^{16} floats, whereas serialization approaches need at least 190 us.

Naos RDMA has lower latency than Skyway, however, Skyway performs better than TCP-Naos for small arrays because of two reasons. First, Naos buffers small objects (less than 256B) to better utilize the network (§3.2). Second, Naos TCP allocates on-heap memory after data arrives, whereas Skyway has all buffers preallocated in our experiments. Both reasons give an advantage to Skyway over Naos TCP for small arrays. For large arrays, Naos TCP provides a 9.1% reduction in latency over Skyway as it incurs fewer data copies.

An array of float is the simplest object graph for graph traversal as it contains a single contiguous object. An array of Point, however, is non-contiguous in memory as this array contains references to objects of class Point, which are 32 bytes each. Nonetheless, Naos provides a 2x and a 4x improve-

ments on average over Java and Kryo for RDMA networks, even with cycle detection enabled (+cycles). *Naos+cycles* benefits from our hot-paths of Algorithm 1 as the JVM tends to collocate objects in memory even for the potentially sparse object graphs. The experiment shows that moderately sparse graphs with small objects are not an issue for Naos.

An array of Pair is even sparser graph than the array of Point, as the class Pair has more references than the class Point. Naos RDMA still achieves the lowest latencies for all sizes. However, with cycle detection, Naos' traversal is slower for long arrays is slower compared to Kryo. The main problem is that Naos sends more data than conventional serializers since it needs to send a JVM header of 16 bytes for each Java object. We conclude that Naos does not always provide lower latency compared to conventional OSD approaches and that its performance depends on sparsity and the number of traversed objects.

A shortcoming of Skyway's and Naos' data format is that they do not compress arrays with references and are forced to send long arrays with (invalid) references, whereas Kryo can encode this information in few bytes. To address this issue, we designed a specialized send call for Naos, namely *NaosIt*, that sends only objects stored in an array. The receiver of such compressed message creates a new array and then fills it with received objects. *NaosIt* reduces the size of communicated data, but requires extra memory allocation on the receiver. Overall, NaosIt provides a small improvement over Naos, as the experiments are performed on 100 Gb/s network. Such compression would be more beneficial for slower networks.

CPU and network costs. To show the key differences between Naos networking and the traditional OSD approaches, Table 3 shows the time breakdown of transferring various data structures and their network cost. Naos as a serialization-free approach always has zero cost for serialization and deserialization. Naos' graph traversal time is included in the send time. The OSD approaches with RDMA has zero receive cost as the data delivered directly to pre-allocated receive buffers by the RNIC. Naos, on the other hand, has non-zero cost as the receive time includes the graph recovery.

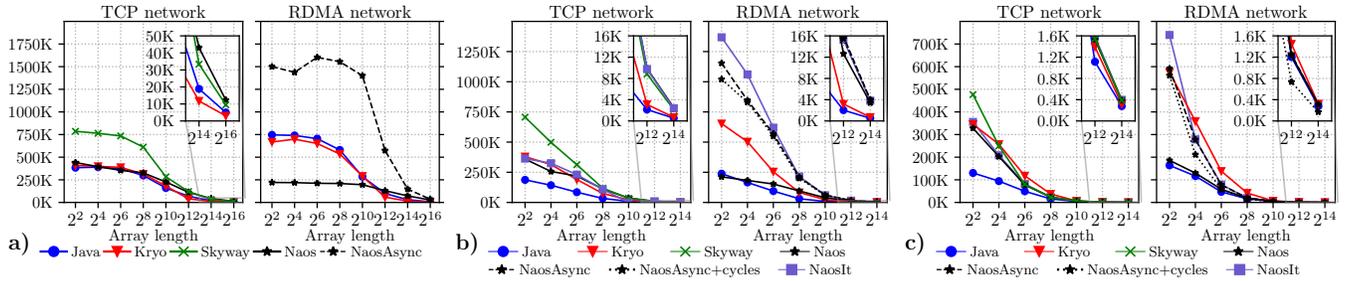


Figure 8: Throughput in objects/sec for a) an array of floats b) an array of Points c) an array of Pairs.

Test	TCP				RDMA				Size (B)
	Ser.	Send	Receive	Deser.	Ser.	Send	Receive	Deser.	
Array of native float with 8192 elements									
Java	15-18	6-9	6-8	17-24	14-17	0-1	0	15-20	32795
Kryo	24-29	7-10	6-8	26-31	24-27	0-1	0	25-86	32772
Skyway	2-3	7-9	7-8	0-1	NA	NA	NA	NA	32792
Naos	0	7-9	16-52	0	0	10-11	0-1	0	32792
Array of class Point with 1024 elements									
Java	109-116	5-7	4-6	94-102	112-119	0-1	0	113-121	14469
Kryo	44-47	4-6	2-3	36-39	43-47	0-1	0	38-41	8132
Skyway	23-26	6-8	6-8	10-11	NA	NA	NA	NA	28696
Naos	0	22-26	27-76	0	0	25-26	13-15	0	28696
NaosIt	0	22-23	28-39	0	0	24-25	15-17	0	24576
Array of class Pair with 1024 elements									
Java	216-664	7-19	10-12	208-222	211-223	0-1	0	217-230	30864
Kryo	135-231	5-10	6-8	79-83	135-148	0-1	0	80-83	18436
Skyway	149-154	9-13	15-31	23-24	NA	NA	NA	NA	61464
Naos	0	161-168	84-137	0	0	199-206	34-39	0	61464
NaosIt	0	159-164	109-138	0	0	200-206	36-40	0	57344
	CPU sender		CPU receiver		CPU sender		CPU receiver		Network

Table 3: CPU time breakdown (in us) and Network cost for transferring arrays. Percentiles 5 and 95 are reported.

Object serialization in TCP experiments takes longer than for RDMA. The difference comes from the fact that in TCP experiments the data is serialized to on-heap buffers, which can be affected by the GC, whereas RDMA requires data to be serialized to off-heap buffers, that are invisible to the GC.

Java and Kryo for RDMA have the same send cost which is the cost of submitting offloaded RDMA request to RNIC. Blocking Naos RDMA has higher cost to send as it needs to wait for a network acknowledgment to finish sending.

For all data types, Naos RDMA shows at least a 2x reduction in CPU time for receiver over Kryo and Java. The main reason is that conventional serialization libraries need to allocate and initialize memory for each received object. Naos does not construct objects and only fixes pointers in the received data. For senders, however, Naos is better at reducing CPU cost for simple graphs such as arrays of floats and points. Note that Naos TCP has a longer receive time as it needs to allocate receive memory, whereas Skyway worked with pre-allocated buffers in our experiments.

The network cost of Naos and Skyway increases with the number of transmitted Java objects. For an array of floats, therefore, the size of the transmitted data is approximately the same for all approaches. On the other hand, for an array of Points or Pairs, the network cost of Naos is about 2x higher in comparison with Java and about 3.5x over Kryo. Kryo has the

lowest network costs as it replaces the class descriptors with integer identifiers significantly compressing object graphs. Naos and Skyway have the same network cost as they have the same data format, but our NaosIt provides a reduction in the network size for array containers.

Throughput. In this experiment, senders continuously send objects to the receiver. For RDMA approaches with serializers, we provide at the sender and the receiver a large number of send and receive buffers to enable asynchronous communication so that the sender can start serializing and sending the next object without the need to wait for the completion of the previous requests.

Figure 8(a) shows that Naos TCP was not able to significantly outperform Skyway for small arrays, as the throughput of Naos was mostly limited by the receive buffer allocation, whereas Skyway, with pre-allocated memory, achieved 750K req/sec. For arrays larger than 2^{12} elements, however, Naos TCP outperforms Skyway as the cost of data copies at the sender overwhelms the cost of memory allocation at the receiver, showing the advantage of our zero-copy design.

The performance of blocking Naos RDMA is bound by the network latency, which prevented the application to send requests at a higher rate. The NaosAsync RDMA, which avoids waiting for an acknowledgment, achieves the highest performance showing the importance of asynchronous communication. For the array of 512 floats, Naos achieves 1600 Kreq/sec, which is a 2x speedup over existing serialization approaches.

Figures 8(b,c) show that the throughput of Naos RDMA was limited by the network bandwidth since NaosIt, that communicates less data, outperforms NaosAsync RDMA. This observation indicates the benefit of our data compression.

The cycle detection decreases the throughput of Naos by less than 3% for moderately sparse graphs. For sparser graphs such as an array of Pairs the slowdown increases to 19%, which is explained by the growth of the Naos' interval tree for cycle detection. Therefore, Naos has lower performance than Kryo, but still outperforms the Java serializer. We think that, in real systems, Naos can be used together with traditional OSD libraries depending on the sparsity of the object graph.

Streaming data transfers. Data processing frameworks such as Spark and Flink rely on data streaming to enable processing of continuous streams of data. The continuous data stream is generated by sending small chunks of data to the

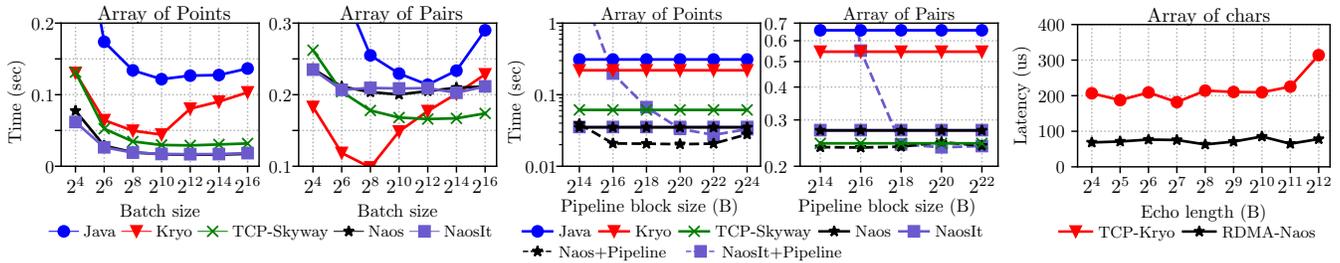


Figure 9: Streaming an array of 2^{20} elements. Figure 10: Pipelining an array of 2^{20} elements. Figure 11: Dubbo RPC latency.

processing nodes. To represent this use-case we implemented the streaming of long data arrays over RDMA networks. Figure 9 shows the streaming time of an array of Points and Pairs with increasing the chunk size.

For the array of Points, Naos RDMA outperforms all serializers for all chunk sizes, and decreases the streaming time of Kryo by 2.1x. For the array of Pairs, Kryo has the highest performance, by sending 256 objects at a time, due to its ability to compress the objects efficiently. For larger chunks, Naos and Skyway take less time than Kryo, since Kryo starts suffering from longer object construction for larger chunks, whereas Skyway and Naos do not need to construct objects.

Even though Skyway and Naos have the same data format, Skyway streamed the array of Pairs faster than Naos. The difference comes from the complexity of Naos’ communication algorithm, leading to the higher CPU cost at the sender (see Table 3). Skyway’s serialization code only copies traversed objects to send buffers, whereas Naos as a communication library needs to take more factors into account: building send lists, RDMA memory registration, and triggering multiple RDMA requests. Naos could employ various modern RDMA techniques for optimized memory accesses [18], which are interesting directions for future research.

Pipelining data transfers. Naos supports pipelining graph traversal with communication on the sender, and pointer fixing with communication on the receiver. Unlike Naos, conventional OSD approaches require an object to be fully serialized before sending it over the network. In this experiment, we show the effect of pipelining for large object graphs by measuring the time of transferring arrays with 2^{20} elements.

The latencies of Java, Kryo, Skyway, and Naos with no pipelining are depicted as straight lines in Figure 10 as they are independent of the pipeline size. Naos with pipelining provides a 20% reduction in latency in comparison with a non-pipelined variant, since the receiver can start pointer recovery earlier. Note that in the previous experiments with streaming large sparse object graphs, Kryo outperformed Naos as it could split the graph into chunks. For inseparable large graphs, however, Naos takes less time even for highly sparse graphs.

Workload	(50:50)	(95:5)	(100:0)
TCP-Kryo	14-24 Kreq/sec	16-24 Kreq/sec	23-25 Kreq/sec
RDMA-Naos	194-266 Kreq/sec	196-266 Kreq/sec	219-281 Kreq/sec

Table 4: Throughput under YCSB workloads with various (read:write) ratios. Percentiles 5 and 95 are reported.

4.2 Accelerating applications with Naos

Naos provides a simple programming interface (see Table 1) hiding all the burden of low-level RDMA communication. In particular, RDMA benchmarks from the previous experiments take only 10 lines for Naos and over 300 lines for the Disni RDMA library. Thus, we believe that it is simple to build systems using Naos. As proof, we have extended Apache Dubbo with Naos communicator, and implemented a Naos-enabled map-reduce framework.

Zero-copy RPC messages with Dubbo. To show that Naos is easy to use, we extended an RPC library Apache Dubbo with the Naos communicator. For that we added a new Naos-enabled communication module that has no serialization module.

In the first experiment we measure the latency of an RPC function that echoes back a Java String. Naos’ performance was compared with the default TCP network library, Mina [3], with Kryo serializer. Naos was deployed with cycle detection. Note that Dubbo besides an RPC arguments also sends an RPC metadata resulting in sending several Java objects. Figure 11 shows that employing Naos RDMA decreases the latency by at least 55% for all tested sizes.

To understand the performance of Naos under a realistic throughput workload, we built a key-value store (KVS) using a Java concurrent hashtable and Dubbo library for communication. We populated the KVS with one million entries of 1 KiB each. We benchmark it under different YCSB [7] workloads. Table 4 shows that Naos RDMA achieves an average speedup of 11x over TCP-Kryo. The speedup comes from the fact that TCP-Kryo was bottlenecked by the CPU, whereas Naos consumes less CPU time to send a KVS request. The experiment shows that Naos can be utilized for KVS workloads as KVS requests and responses have low sparsity.

In comparison with microbenchmarks (§4.1), the performance difference between Naos and Kryo is much higher for the current workload than for the microbenchmarks, where Kryo’s performance was measured after JIT compilation that

Test	LiveJournal [4] (2 nodes)				Orkut [35] (3 nodes)			
	TCP		RDMA		TCP		RDMA	
	Total	Stage	Total	Stage	Total	Stage	Total	Stage
Java	413.46	3.67-4.04	406.87	3.53-3.99	364.53	3.20-3.38	365.29	3.10-3.44
Kryo	410.94	3.62-4.06	411.33	3.63-3.99	357.58	2.98-3.47	354.06	2.91-3.42
Skyway	394.32	3.52-3.77	NA	NA	350.48	3.07-3.24	NA	NA
Naos	393.05	3.54-3.76	395.05	3.59-3.70	342.06	2.85-3.32	343.00	2.84-3.35
NaosIt	394.49	3.56-3.77	386.01	3.48-3.69	345.94	2.82-3.45	333.16	2.72-3.24
Skyway [†]	386.31	3.54-3.78	NA	NA	340.82	2.95-3.30	NA	NA
Naos [†]	373.04	3.27-3.72	369.10	3.16-3.63	331.31	2.86-3.22	335.50	2.85-3.22

[†] PageRank with sparsity-aware implementation.

Table 5: Total and per stage processing times in seconds for 100 iterations of PageRank algorithm. Percentiles 5 and 95 are reported for PageRank iterations.

significantly improved its performance for repetitive sending of the same object. Since Naos does not depend on Java runtime optimizations, it can achieve much higher performance than Kryo for dynamic workloads.

Improving Data Processing Applications. We could not integrate Naos into Spark as its shuffle module is designed to communicate files with serialized objects. Integration of Naos would require a substantial redesign of Spark’s code base. Therefore, we implemented our own map-reduce framework that takes advantage of Naos. Our framework supports all discussed serializers including Skyway and also offers RDMA networking with Disni. It was designed to resemble Spark but perform shuffle completely in-memory.

We evaluate the OSD approaches by running PageRank on real-world graphs as input: LiveJournal [4] and Orkut [35]. The LiveJournal dataset was processed with two shuffle workers and Orkut with three shuffle workers. Naos was deployed with 256 KiB pipelining and without cycle detection. We report total runtime including data loading and 5 and 95 percentiles for processing a single Pagerank iteration. We also provide two implementations of PageRank: the first one follows conventional design where each score update is a class of 32 bytes; the second implementation was designed to communicate dense contiguous score updates, thereby reducing sparsity of communicated shuffle blocks.

Table 5 shows the lowest runtime was achieved by Naos and Skyway for the first implementation. A side effect of Naos and Skyway is that, after receiving, objects are always contiguous in memory, thereby improving data locality. As a result, an application can process such contiguous objects faster as fewer memory pages need to be fetched. Overall, Naos TCP performs approximately as Skyway, but NaosIt RDMA provides 2.1% and 4.8% improvement over Skyway for LiveJournal and Orkut, respectively. The experiment shows that zero-transformation approaches for OSD can reduce processing time for data-processing workloads.

The sparsity-aware implementation provides an additional 4% reduction in runtimes, showing that applications need to take Naos’ limitations into consideration to achieve the highest performance. Thus, Naos could be used in combination with works on data sparsity reduction for JVMs [5, 33, 34].

5 Discussion and Future work

The role of RDMA. RDMA helps Naos to remove potential copies induced by the TCP stack. Application-wise, Naos is zero-copy for both TCP and RDMA networks, unlike Skyway. On the other hand, for trivial graphs, Skyway and Naos TCP use almost identical algorithms for the receiver, as they both receive objects with zero-copy and only fix the class reference. However, since Naos TCP does not pre-allocate memory for receiving, its performance could be bound by memory allocation. It is possible to modify Naos TCP to pre-allocated buffers as Skyway and RDMA Naos do, removing the bottleneck. In this work, however, we focus on the RDMA implementation of Naos.

SmartNICs. In Naos, a sender cannot modify its on-heap memory before sending. Therefore, a receiver has to employ a complex pointer recovery algorithm, whereas Skyway can pre-process buffers before sending them to help the receiver to recover objects faster. We believe that such a feature is better implemented at the SmartNIC level that would fix the pointers on the fly before writing the data to DRAM. For example, since a Naos’ RDMA sender already knows the destination addresses of the objects, either the SmartNIC at the sender or at the receiver could fix the object pointers. The class pointers could be fixed by storing class translation tables in the SmartNIC.

6 Conclusions

We have presented Naos, a JVM communication library that enables transferring objects directly from one heap to another over the network with minimal CPU involvement and zero-copy. We demonstrated that existing OSD techniques are bound to CPU and that, as networks get faster, they will become the bottleneck of distributed systems. Naos completely avoids the need to serialize and deserialize objects for data transfers, with the corresponding performance advantages. Naos provides a simple API that simplifies the use of RDMA from JVM-based applications. Our evaluation shows that Naos outperforms all existing OSD approaches for moderately sparse object graphs.

7 Acknowledgement.

We would like to thank our shepherd, Khanh Nguyen, and the anonymous reviewers for their suggestions and help to improve the paper. This research was supported by ETH Zurich, and by Microsoft Research through its Swiss Joint Research Centre. The project also received funding from the European Research Council under the European Union Horizon 2020 programme (grant agreement DAPP, No. 678880). Rodrigo Bruno’s research was supported in part by grants from Oracle Labs and SBB.

References

- [1] Oracle Corporation and/or its affiliates. JDK 11, 2019. <https://openjdk.java.net/projects/jdk/11/>.
- [2] Dubbo Apache. Apache Dubbo Project, 2018. <https://dubbo.apache.org>.
- [3] MINA Apache. Apache MINA Project, 2009. <https://mina.apache.org>.
- [4] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD'06, pages 44–54. Association for Computing Machinery, 2006.
- [5] Rodrigo Bruno, Vojin Jovanovic, Christian Wimmer, and Gustavo Alonso. Compiler-Assisted Object Inlining with Value Fields. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI'21. Association for Computing Machinery, 2021.
- [6] 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*, 36(4), 2015.
- [7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghuram Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC'10, pages 143–154. Association for Computing Machinery, 2010.
- [8] Aleksandar Dragojević, Dushyant 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. USENIX Association, 2014.
- [9] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. Shenandoah: An Open-Source Concurrent Compacting Garbage Collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ'16. Association for Computing Machinery, 2016.
- [10] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In Pierre America, editor, *ECOOP'91 European Conference on Object-Oriented Programming*, pages 21–38. Springer, 1991.
- [11] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference*, USENIX ATC'10. USENIX Association, 2010.
- [12] Google Inc. Protocol buffers: Google's data interchange format, 2008. <https://github.com/protocolbuffers/protobuf>.
- [13] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W. Lee. A specialized architecture for object serialization with applications to big data analytics. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA'20, page 322–334. IEEE Press, 2020.
- [14] 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. USENIX Association, 2016.
- [15] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a Warehouse-Scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA'15, pages 158–169. Association for Computing Machinery, 2015.
- [16] Sean Leary. JSON-java, 2015. <https://github.com/stleary/JSON-java>.
- [17] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafir. Page Fault Support for Network Controllers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'17, pages 449–466. Association for Computing Machinery, 2017.
- [18] Mellanox Technologies Ltd. Optimized memory access, 2020. <https://docs.mellanox.com/display/MLNXOFEDv492240/Optimized+Memory+Access>.
- [19] X. Lu, D. Shankar, S. Gugnani, and D. K. Panda. High-performance design of Apache Spark with RDMA and its benefits on various workloads. In *2016 IEEE International Conference on Big Data*, BigData'16, pages 253–262, 2016.
- [20] Frank Mietke, R. Baumgartl, R. Rex, Torsten Mehlan, Torsten Hoeffler, and Wolfgang Rehm. Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack. In *Proceedings of Euro-Par 2006*

- Parallel Processing*, pages 124–133. Springer-Verlag Berlin, 2006.
- [21] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. Skyway: Connecting Managed Heaps in Distributed Big Data Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS’18, pages 56–69. Association for Computing Machinery, 2018.
- [22] 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. USENIX Association, 2015.
- [23] Marius Poke and Torsten Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC’15, pages 107–118. Association for Computing Machinery, 2015.
- [24] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus Prime: Accelerating Data Transformation in Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS’20, pages 1203–1216. Association for Computing Machinery, 2020.
- [25] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMark: Bypassing RDMA Security Mechanisms. In *30th USENIX Security Symposium*, USENIX Security’21. USENIX Association, 2021.
- [26] Esoteric Software. Kryo - object graph serialization framework for Java, 2008. <https://github.com/EsotericSoftware/kryo>.
- [27] Patrick Stuedi, Bernard Metzler, and Animesh Trivedi. JVerbs: Ultra-Low Latency for Data Center Applications. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SoCC’13. Association for Computing Machinery, 2013.
- [28] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. DaRPC: Data Center RPC. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC’14, pages 1–13. Association for Computing Machinery, 2014.
- [29] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. Unification of Temporary Storage in the NodeKernel Architecture. In *2019 USENIX Annual Technical Conference*, USENIX ATC’19, pages 767–782. USENIX Association, 2019.
- [30] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. CoRM: Compactable Remote Memory over RDMA. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, SIGMOD’21. Association for Computing Machinery, 2021.
- [31] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. sRDMA – Efficient NIC-based Authentication and Encryption for Remote Direct Memory Access. In *2020 USENIX Annual Technical Conference*, USENIX ATC’20, pages 691–704. USENIX Association, 2020.
- [32] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. Morpheus: Creating application objects efficiently for heterogeneous computing. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA’16, pages 53–65. IEEE Press, 2016.
- [33] Christian Wimmer and Hanspeter Mössenböck. Automatic Feedback-Directed Object Inlining in the Java Hotspot™ Virtual Machine. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE’07, pages 12–21. Association for Computing Machinery, 2007.
- [34] Christian Wimmer and Hanspeter Mössenböck. Automatic Array Inlining in Java Virtual Machines. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO’08, pages 14–23, New York, NY, USA, 2008. Association for Computing Machinery.
- [35] Jaewon Yang and Jure Leskovec. Defining and Evaluating Network Communities Based on Ground-Truth. *Knowl. Inf. Syst.*, 42(1):181–213, 2015.
- [36] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud’10, page 10. USENIX Association, 2010.

One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory

Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, Yu Hua[†]

Huawei Cloud

[†]*Huazhong University of Science and Technology*

Abstract

Memory disaggregation is a promising technique in datacenters with the benefit of improving resource utilization, failure isolation, and elasticity. Hashing indexes have been widely used to provide fast lookup services in distributed memory systems. However, traditional hashing indexes become inefficient for disaggregated memory since the computing power in the memory pool is too weak to execute complex index requests. To provide efficient indexing services in disaggregated memory scenarios, this paper proposes RACE hashing, a one-sided **RDMA-Conscious Extendible** hashing index with lock-free remote concurrency control and efficient remote resizing. RACE hashing enables all index operations to be efficiently executed by using only one-sided RDMA verbs without involving any compute resource in the memory pool. To support remote concurrent access with high performance, RACE hashing leverages a lock-free remote concurrency control scheme to enable different clients to concurrently operate the same hashing index in the memory pool in a lock-free manner. To resize the hash table with low overheads, RACE hashing leverages an extendible remote resizing scheme to reduce extra RDMA accesses caused by extendible resizing and allow concurrent request execution during resizing. Extensive experimental results demonstrate that RACE hashing outperforms state-of-the-art distributed in-memory hashing indexes by 1.4 – 13.7× in YCSB hybrid workloads.

1 Introduction

Memory disaggregation, which has attracted extensive attentions from both industry, e.g., HP’s The Machine [20] and Intel RSD [21], and academia [6, 16, 18, 28, 29, 38], decouples the traditional monolithic compute and memory resources in datacenters and forms independent compute and memory resource pools. Due to resource pooling and independent hardware deployments, disaggregated memory enjoys the benefits of improvements on resource utilization, failure isolation, and elasticity [5, 42]. In the disaggregated memory architecture, compute blades run applications with only a small amount of memory as cache. In contrast, the memory pool stores application data with weak computing power. Due to not involving the compute resources in the memory pool, fast one-sided

RDMA networks generally serve for data accesses from the compute blades to the memory pool.

Distributed in-memory hashing indexes have become one of the fundamental building blocks in many datacenter applications, such as databases [23, 27, 45] and key-value stores [2, 3, 25]. With the increasing popularity of RDMA in modern datacenters, RDMA-search-friendly (RSF) hashing indexes have been intensively studied, e.g., FaRM hopscotch hashing [13], Pilaf cuckoo hashing [31], and DrTM cluster hashing [44]. These RSF indexes execute search requests by using one-sided RDMA `READS` to fetch data from remote memory without involving remote CPUs. In contrast, insertion, deletion, and update (IDU) requests are sent to the remote CPUs, which locally execute them. However, this mechanism fails to work in the new disaggregated memory architecture, since the computing power in the memory pool is too weak to execute the aforesaid complex IDU requests. In fact, in these RSF hashing indexes, IDU requests can be executed in the compute blades by using one-sided RDMA `WRITE` and `ATOMIC` verbs to operate on remote data. However, we observe that executing IDU requests using one-sided RDMA verbs in existing RSF hashing indexes incurs significant performance degradation, due to a large number of network round-trips and concurrent access conflicts. In a nutshell, it is non-trivial to design an efficient hashing index for disaggregated memory due to the following challenges:

- *Many remote reads&writes for handling hash collisions.*

In order to handle hash collisions, existing hashing schemes incur significant data movement overheads to make room for newly inserted items, e.g., hopscotch hashing [19] and cuckoo hashing [36]. These data movements are executed by many remote reads and writes in the disaggregated memory, which significantly decrease the performance of hashing indexes, since each remote read or write produces one RDMA network round-trip.

- *Concurrency control for remote access.* To handle conflicts of concurrent accesses, lock-based techniques have been widely used in hashing indexes [15, 26]. Locks have low overhead for local hashing indexes, due to nanosecond-level latency for local execution. However, when using locks for hashing indexes in disaggregated memory scenarios, remote locking has to be implemented using RDMA `ATOMIC` verbs

with microsecond-level latency, thus incurring high overheads and increasing the waiting delay when lock contention occurs. Especially for the hashing indexes with excessive data movement, multiple locks are acquired before moving data, which exacerbates the lock contention.

- *Tricky remote resizing of hash tables.* When a hash table is full, resizing is inevitable for increasing its size. Conventional full-table resizing needs to move all key-value items from an old hash table to a new one. Extendible resizing [14,33] reduces the number of moved items during resizing, at the cost of one extra RDMA READ due to the need of first accessing the directory of the hash table. Moreover, during resizing, it is challenging to concurrently access the hash table.

To address the above challenges, we propose RACE hashing, to the best of our knowledge, the first hashing index designed for disaggregated memory which fully relies on one-sided RDMA verbs to efficiently execute all index requests. To reduce the performance influence of resizing, RACE hashing leverages the extendible resizing, and hence a RACE hash table consists of multiple subtables and a directory which is used to index subtables. The subtable structure is designed to be one-sided RDMA-conscious (RAC), achieving that all index requests (including search, insertion, deletion, and update) can be executed using only one-sided RDMA verbs while having a constant-scale time complexity in the worst case, and therefore delivering high performance. To improve the performance of remote concurrency, RACE hashing leverages a lock-free remote concurrency control scheme for the RAC hash subtable, which achieves that all index requests except failed insertions are concurrently executed in a lock-free manner. Moreover, to reduce the performance penalty from extendible resizing, RACE hashing caches the directory at the client side (a client is a CPU blade), and therefore eliminates the RDMA access to the directory. Nevertheless, since the directory in the client cache becomes stale when the hash table is resized, accessing the hash table via a stale directory cache may obtain incorrect or inconsistent results. RACE hashing presents a simple yet efficient stale-read scheme to guarantee the correctness of accessed data and allow current request execution during resizing.

Specifically, this paper makes the following contributions:

- *One-sided RDMA-conscious table structure.* We present a RAC subtable structure that is both RDMA-search-friendly and RDMA-IDU-friendly. All index requests are executed by using only one-sided RDMA verbs with constant worst-case time complexity. IDU requests do not cause any extra data movement.

- *Lock-free remote concurrency control.* We design lock-free remote concurrent algorithms for RACE hashing to enable all requests except failed insertions to be concurrently executed without locking.

- *Extendible remote resizing.* We present a stale-read client directory cache scheme to reduce one extra RDMA READ for remote directory lookups and guarantee request execution

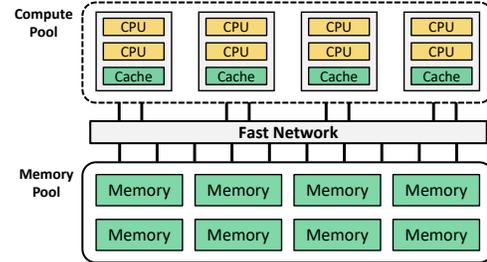


Figure 1: Architecture for disaggregated memory.

correctness when using the stale directory cache. We also achieve concurrent access to the subtable that is being resized.

- *Implementation and evaluation.* We have implemented the RACE hashing and evaluated its performance. Extensive experimental results demonstrate that RACE hashing outperforms state-of-the-art distributed in-memory hashing indexes by up to 13.7× in YCSB [11] hybrid workloads.

2 Background and Motivation

2.1 Disaggregated Memory

In a general disaggregated memory architecture in datacenters [6, 18, 28, 29, 42], different types of resources are separated into pools, e.g., a compute pool and a memory pool, as shown in Figure 1. Each pool is managed and scaled independently as well as failure-isolated. The compute pool consists of many CPU blades, each of which retains a small amount of memory as the local cache for the memory pool. The memory pool includes many memory blades that can be DRAM or persistent memory DIMMs, RNICs, and controllers (the RNIC and controller can be the same entity). The RNIC and controller have low-power processing units used only for interconnection. The communication between compute and memory pools leverages fast remote-access interconnect techniques, such as one-sided RDMA, Omni-path [8], or Gen-Z [1]. The interfaces that the memory pool provides for the compute pool include READ, WRITE, ALLOC, and FREE for variable-size memory blocks, as well as ATOMIC operations, e.g., compare-and-swap (CAS) and fetch-and-add (FAA). We assume ALLOC and FREE interfaces are implemented in the RNICs or controllers of the memory pool [1, 42]. Without loss of generality, the rest of this paper considers using one-sided RDMA for the interconnect in the disaggregated memory architecture, i.e., compute blades access the memory pool using RDMA READ, WRITE, and ATOMIC verbs.

2.2 RDMA-search-friendly Hashing Index

In this subsection, we first present existing RDMA-search-friendly (RSF) hashing indexes and then analyze their performance on disaggregated memory.

2.2.1 Existing Hashing Schemes

With the wide use of RDMA in modern datacenters, RSF hashing indexes have been intensively studied [13, 31, 44]. All

these RSF hashing indexes are designed for the datacenter architecture with monolithic servers. Clients execute search requests by using RDMA READs to fetch data from remote memory without involving remote CPUs. In contrast, IDU requests are sent to the remote servers and executed using the remote CPUs. We review each of these hashing indexes in detail as follows.

Pilaf Cuckoo Hashing: Pilaf [31] proposes a 3-way cuckoo hashing that uses 3 orthogonal hash functions to compute 3 different hash buckets for each key. When executing a key Search, the client first reads one of its 3 corresponding hash buckets using an RDMA READ. If the key does not exist in the first bucket, the client then reads the second hash bucket. Upon not finding the key in the second bucket, the client reads the third hash bucket. For Insertion requests, the client sends them to the server and the server CPUs handle them locally. An insertion may iteratively evict existing key-value items in the cuckoo hash table to their alternate locations. This mechanism incurs an inconsistency problem in which a search request executed by the client may miss the key when the server is handling its eviction. To address this problem, the server first calculates all affected buckets (called a cuckoo path [26]) before moving keys. The server then moves each key to its alternate location starting from the last affected bucket in the cuckoo path.

FaRM Hopscotch Hashing: FaRM [13] proposes a chained associative hopscotch hashing in which each bucket has a neighborhood that includes the bucket itself and its following bucket. Each bucket has multiple slots and each key is stored in the neighborhood of the bucket that the key is hashed to. For an Insertion that is also handled at the server side, the hopscotch hashing tries to find an empty slot in the neighborhood of the key's hash bucket. If found, the empty slot stores the item. Otherwise, the hopscotch hashing continues to find an empty slot forward by executing a *linear probe*. If finding an empty slot, the hopscotch hashing tries to iteratively displace items to *move* the empty slot towards the neighborhood. If there is no empty slot or the movement fails, the hopscotch hashing stores the item in the bucket list linked to the key's hash bucket. When executing a Search, the client reads the neighborhood of the key's hash bucket, i.e., two adjacent buckets, using an RDMA READ. Upon not finding the key, the client further traverses the linked buckets. Note that traversing each bucket needs an RDMA READ.

DrTM Cluster Hashing: Cluster hashing proposed in DrTM [44] is a chained hashing with associativity, in which reading and writing key-value items use RDMA READs and WRITEs and insertions and deletions to the hash table are shipped to the server for local execution. To insert a new key, the cluster hashing tries to find an empty slot in the key's hash bucket and the bucket list linked to the key's hash bucket. If there is no empty slot, the cluster hashing adds a new bucket in the bucket list to store the inserted key-value item. For a Search request, the client reads the key's hash bucket using

an RDMA READ. Upon not finding the key, the client further traverses the linked buckets one by one.

2.2.2 Performance on Disaggregated Memory

To the best of our knowledge, there is no existing hashing index specifically designed for disaggregated memory. As a first step, we analyze the performance of using the above RSF hashing indexes in disaggregated memory. Due to the absence of computing power in the memory pool to execute their IDU requests, we consider implementing the IDU requests with one-sided RDMA verbs.

For *Pilaf cuckoo hashing* [31], to insert a key-value item, the client needs to execute eviction operations when the hash table is in a high load factor. Specifically, based on the state-of-the-art concurrent cuckoo hashing algorithm [26], the client first calculates a cuckoo path and locks all buckets in the path using RDMA CASEs. The client then uses RDMA WRITEs to iteratively evict key-value items in the cuckoo path. A cuckoo path may include tens or hundreds of buckets [15]. Thus an insertion is executed by using a large number of RDMA CASEs and WRITEs, delivering poor insertion performance and also decreasing the performance of other search requests due to the use of a large number of locks.

For *FaRM hopscotch hashing* [13], to insert a key-value item, the client needs to linearly probe buckets in the hash table using RDMA READs until finding an empty slot. When the hash table is in a high load factor, inserting a key may need to read the entire hash table to the client until finding an empty slot. After finding an empty slot, moving the empty slot toward the neighborhood of the inserted key is also complex and expensive, due to locking multiple buckets in the movement path and using multiple RDMA WRITEs to move items. Moreover, if there is no empty slot or the movement fails, the operation of adding linked buckets is also expensive.

For *DrTM cluster hashing* [44], to insert a key-value item, the client needs to traverse buckets in its corresponding bucket list one by one until finding an empty slot. Traversing each bucket needs an RDMA READ. If there is an empty slot in these buckets, the client inserts the item using an RDMA WRITE. Otherwise, the client adds a new overflow bucket to the bucket list. Before modifying the bucket list, the client needs to lock the bucket list to prevent other clients from inserting duplicate keys or freeing buckets. Thus an insertion executes operations including traversing the bucket list, locking/unlocking, allocating memory for a new bucket and the new item, linking the new bucket, and writing the new item, resulting in many RDMA READs, WRITEs, and CASEs. The operations of allocating overflow buckets in the cluster hashing are more frequent than in FaRM hopscotch hashing, since the cluster hashing has a weaker ability to deal with hash collisions in the main hash table. Moreover, the deletion requests are also complex in the structure of linked bucket lists, due to the need of moving items from buckets at the list tail towards ones at the list head to fill empty slots and recycling

tail buckets for higher performance and space utilization [13].

In summary, these RDMA-search-friendly hashing indexes become RDMA-IDU-unfriendly for disaggregated memory since IDU requests incur a large number of RDMA operations to deal with hash collisions and concurrency control. Our paper proposes RACE hashing which is *both RDMA-search-friendly and RDMA-IDU-friendly* while efficiently dealing with hash collisions and concurrency control as presented in Section 3. The performance is also verified in Section 4.

2.3 Resizing Hash Tables

When a hash table is full, i.e., an insertion failure occurs or its load factor reaches a threshold, the hash table needs to be resized by expanding its capacity. In general, there are two kinds of resizing mechanisms including full-table resizing and extendible resizing [14, 33].

To expand a hash table, the *full-table resizing* mechanism allocates a new hash table whose size is larger than the old one, e.g., double the size, and then iteratively moves each key-value item from the old hash table to the new one. The full-table resizing is expensive due to moving all items.

In the *extendible resizing*, a resizing operation only needs to move partial items. Specifically, the hash table using extendible resizing includes multiple subtables and there is a directory to index these subtables as shown in Figure 2a. For a 64-bit hash value, M bits are used by the directory to locate a subtable (we use the last M bits as an example, i.e., suffix) and the remaining $(64 - M)$ bits are used to locate target buckets within the subtable. The number of suffix bits currently used by the directory is called *global depth (GD)* ($GD \leq M$). Thus the directory has 2^{GD} entries that correspond to at most 2^{GD} subtables. Each subtable has a *local depth (LD)* ($LD \leq GD$) that indicates the number of suffix bits used by the subtable.

When a subtable is full, we split the subtable into two by adding a new subtable. As shown in Figures 2a and 2b, when the subtable with the suffix “1” is full, it is split into Subtables “01” and “11”. The resizing mechanism moves the key with suffix “11” from Subtable “01” to Subtable “11” and changes their *LDs* to 2. When a subtable is full and its *LD* is equal to the *GD*, we grow the directory by doubling its size, as shown in Figures 2b and 2c. The full subtable is split into two ones. Except for the directory entry that the added new subtable corresponds to, other new directory entries point to their corresponding original subtables. After resizing the directory, search requests use the new *GD* to locate their corresponding subtables. In summary, by performing extendible resizing, when a subtable is full, we only need to resize this single subtable without affecting key-value items in other subtables. Therefore, RACE hashing uses the extendible resizing.

Nevertheless, there are two challenges when using extendible resizing in disaggregated memory. First, compared with the full-table resizing, the extendible resizing incurs one extra memory access for each search request, due to the need of first querying the directory to obtain the address of the tar-

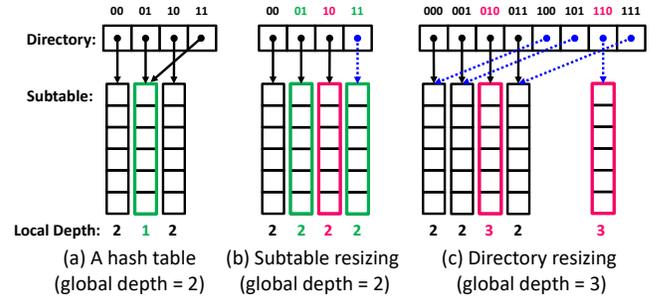


Figure 2: A hash table with extendible resizing.

get subtable before accessing the subtable. One extra memory access has little impact on the performance of a local hash table, due to fast local memory access. However, in the disaggregated memory, the one extra memory access *produces one more RDMA round-trip*, significantly decreasing the search performance. Second, as there is no powerful compute resource in the disaggregated memory to execute the complex resizing, the resizing has to be triggered and executed by a remote client, i.e., a CPU blade, which is different from the traditional resizing mechanism that is always executed by local CPUs. When a client is performing the resizing, other clients do not know about its occurrence. Therefore, we have to *deal with concurrent access to the hash table during resizing*.

3 RACE Hashing

3.1 Overview

Figure 3 shows the overall architecture of RACE hashing for disaggregated memory. The RACE hash table is stored in the memory pool. Clients in the compute pool operate the hash table using one-sided RDMA verbs. To alleviate the performance influence of resizing, RACE hashing leverages the extendible resizing and hence the hash table consists of multiple subtables and a directory. In order to reduce the extra RDMA *READ* for accessing the remote directory, RACE hashing leverages a directory cache¹ in the client. Each client maintains a local cache to store only the directory of the RACE hash table. Thus a client can access the directory using a local memory access rather than a remote RDMA *READ*, and use RDMA verbs to access only the subtable. We present the design of an *RDMA-conscious (RAC) hash subtable structure* in Section 3.2, i.e., the RAC hash subtable, in which all index requests are executed by using only one-sided RDMA verbs while having constant worst-case time complexity. We then present a *lock-free remote concurrency control scheme* in Section 3.3 for the RAC hash subtable, achieving that index requests including search, insertion, deletion, and update are concurrently executed in a lock-free way. Moreover, caching the directory in clients causes data inconsistency issues between the directories in the memory pool and client caches.

¹The memory overhead of the cache is small since the directory generally has at most hundreds of entries and each entry has only several bytes.

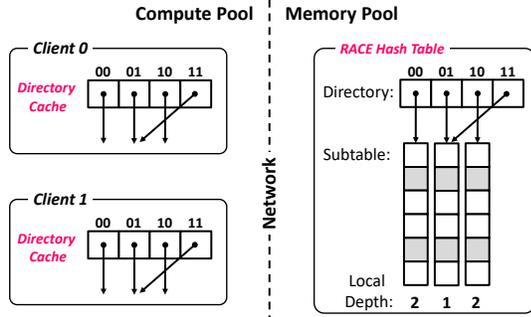


Figure 3: The overall architecture of RACE hashing in disaggregated memory. (The entire RACE hash table is stored in the memory pool. Clients in the compute pool store the directory of the hash table in their local caches and access subtables only using one-sided RDMA verbs.)

Therefore, we finally present a client directory cache with stale reads scheme in Section 3.4 to address the inconsistency issue at low overhead.

3.2 The RAC Hash Subtable Structure

In disaggregated memory scenarios, the challenge of designing a RAC hash subtable structure stems from minimizing the number of remote RDMA operations for IDU requests while keeping high memory efficiency and Search performance. To achieve this goal, we design the RAC hash subtable that does not allow any movement operations, evictions, or bucket chaining to handle hash collisions, since these operations incur a large number of remote writes as presented in Section 2.2. Instead, the RAC hash subtable uses three major design choices, including associativity, two choices, and overflow colocation, for addressing hash collisions and thus achieves a constant worst-case time complexity for all index requests.

1) Associativity. With associativity, each bucket has multiple slots, being capable of storing multiple key-value items. K -way associativity means that each bucket has K slots. Associativity is friendly for one-sided RDMA operations since multiple items within one bucket can be read together in one RDMA READ. Figure 4 shows a RAC hash subtable with 4-way associativity.

2) Two Choices. Based on the theory of “the power of two choices” [32], enabling each key to have two choices for its storage location can achieve a good load balance among buckets, effectively handling hash collisions. Hence, the RAC subtable uses two independent hash functions, $h_1()$ and $h_2()$, to compute two hash locations for each key, as shown in Figure 4. By efficiently combining associativity with two choices, the RAC subtable inserts a new item into the less-loaded bucket between its two hash locations. Note that, according to Mitzenmacher’s observations [32], two choices achieve exponential improvements over one choice for the efficiency of load balancing, while three choices only have a constant factor improvement than two choices. In disaggregated memory, three choices incur one more bucket access (i.e., one more

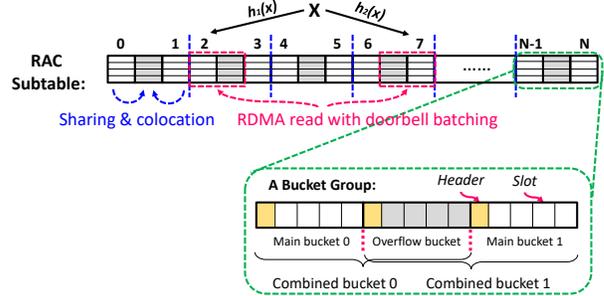


Figure 4: The RAC hash subtable structure (4-way associativity as an example).

RDMA READ) than two choices. Therefore, unlike Pilaf [31], which uses three choices, we use two choices in our design.

3) Overflow Colocation. The overflow sharing technique [46] enables an overflow bucket (or called standby bucket) to be shared by the other two main buckets to store conflicting items for better load balancing. However, overflow buckets are discrete from their main buckets [46], incurring extra bucket accesses, which performs worse for disaggregated memory, due to extra RDMA READs. To address this problem, we propose an overflow colocation scheme to store the overflow buckets adjoining with their main buckets. As shown in Figure 4, three continuous buckets are considered as a group, in which the first and last buckets are main buckets that can be addressable by the hash functions. The middle bucket is an overflow bucket that cannot be addressable by the hash functions and is shared by the first and last buckets to store their conflicting items. By doing so, one RDMA READ can fetch one main bucket and its overflow bucket together, thus reducing the number of RDMA READs.

Putting it all together, the structure of a RAC hash table is shown in Figure 4. A RAC hash subtable is a one-dimensional bucket array stored in a continuous memory space. Each bucket is K -way associative and a bucket group includes three continuous buckets, i.e., two main buckets and a shared overflow bucket. The combination of a main bucket and its overflow bucket is called a combined bucket. For each key, we compute two hash locations that are respectively in two different bucket groups. The structure of the RAC hash subtable is simple yet efficient for disaggregated memory, having the following strengths:

- **RDMA-IDU friendly:** As each key only involves two combined buckets, IDU requests only need to operate within the two combined buckets without moving/evicting items from/to other buckets or linking new buckets, having constant worst-case time complexity while being RDMA-friendly.
- **RDMA-search friendly:** A search request only issues two RDMA READs, each of which fetches one combined bucket. More importantly, the two RDMA READs can be issued in parallel to reduce the request latency, unlike cluster hashing [44] in which issuing the next RDMA READ has to wait for the return of the previous one to traverse the linked buckets.

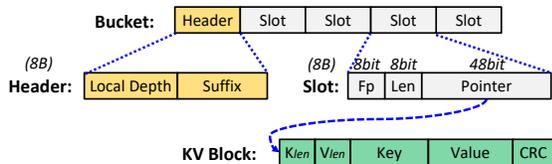


Figure 5: The structure of a bucket.

Moreover, by using doorbell batching [22] that is an RDMA-optimized technique to read multiple disjoint memory regions within one RDMA round-trip time (RTT), we package the two RDMA READ operations into one. Therefore, the search latency in the RAC subtable is one RTT rather than two ones.

- *High memory efficiency:* By combining associativity, two choices, and overflow colocation, the RAC hash subtable enables items to be more evenly distributed among buckets and thus efficiently handles hash collisions to achieve a good load balance. It hence achieves a high load factor of up to 90% (with 7-way associativity) as evaluated in Section 4.2.1.

3.3 Lock-free Remote Concurrency Control

Lock-based techniques have been widely used in existing hashing indexes within a single machine for concurrency control [15, 26]. Nevertheless, for disaggregated memory, all requests are executed by using one-sided RDMA verbs, which results in non-trivial challenges for handling concurrent access conflicts. This is because remote locking implemented by using microsecond-level-latency RDMA CAS incurs much higher overheads, compared with nanosecond-level-latency local locking, and each locking or unlocking operation requires an RDMA round-trip. In order to deliver high concurrent performance, we propose a *lock-free remote concurrency control scheme* for RACE hashing which achieves that all index requests, except failed insertions, become lock-free. A failed insertion triggers a subtable resizing and needs to acquire the resizing lock as presented in Section 3.4.2.

Bucket Structure. In RACE hashing, to support variable-length keys and values, full key-value items are stored outside the hash table like existing hashing indexes [10, 31, 44]. The pointers to full key-value items are stored inside the hash table. The structure of each bucket in the RAC hash subtable is shown in Figure 5. A bucket consists of a header and multiple slots. The header is used for hash table resizing and will be introduced in Section 3.4.1. Each slot corresponds to a key-value item. To support lock-free remote concurrent access, a slot is 8B, i.e., the maximum size of an RDMA CAS, and composed of a fingerprint (8 bits), a key-value length (8 bits), and a pointer (48 bits). A fingerprint (Fp) is the 8-bit hash of a key. Based on the analysis of existing work [15], an 8-bit fingerprint is enough to achieve a very low false positive (a false positive means that different keys in a bucket have the same fingerprint). Moreover, before reading a full key-value item using an RDMA READ, we need to know the size of the item. Therefore, we store the length (Len) of the key-

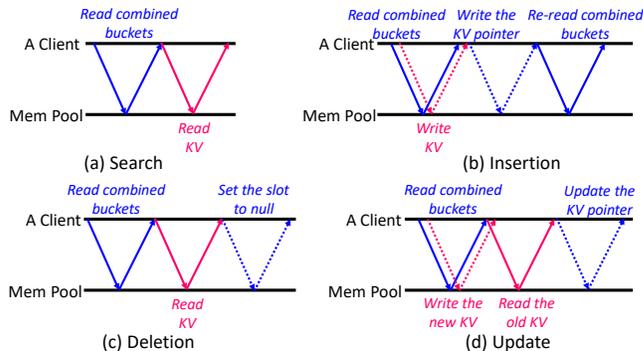


Figure 6: The main workflows of lock-free search, insertion, deletion and update. (The blue lines mean accessing the hash table and the red lines mean accessing the key-value blocks. The solid lines mean RDMA READ round-trips and the dotted lines mean RDMA WRITE/ATOMIC round-trips.)

value block in the slot. The length is 8-bit and the length unit is 64B². Thus the length of a key-value block is always multiple of 64B and the maximum length of a key-value block is $2^8 * 64B = 16KB$, which covers most application scenarios for current key-value stores since small key-values dominate in them [7]. When a key-value item is larger than 16KB, which in fact rarely occurs, we store the remaining item content beyond 16KB in the second key-value block and link the second block to the first one. The respective lengths of the key and value (i.e., K_{len} and V_{len}) are stored in the head of the key-value block. The pointer in a slot consumes 48 bits like the x86_64 system [10, 34, 43]. A null pointer means the slot is empty. Based on the bucket structure, we present lock-free search/insertion/deletion/update operations below.

Lock-free Insertion. To insert a key-value item, the client uses doorbell batching to read two combined buckets that the key corresponds to. At the same time, the client writes the key-value block³ in the memory pool. Therefore, reading buckets and writing the key-value block are executed in parallel, as shown in Figure 6b. Once receiving the combined buckets, the client looks for an empty slot in the order of main buckets first and overflow buckets second, as presented in Section 3.2. If an empty slot is found, the client uses an RDMA CAS to write the pointer of the key-value block into it. Otherwise, the hash table resizing is triggered, as presented in Section 3.4.

In rare cases, clients may concurrently insert duplicate keys into the hash table, since RDMA ATOMIC verbs only ensure the 8B atomicity. For example, Client 1 and Client 2 try to insert the same key K . As each key corresponds to two combined buckets in RACE hashing, it may occur that Client 1 selects one empty slot in Combined Bucket 0 to insert K and Client 2 selects one empty slot in Combined Bucket 1 to insert K . In this case, two duplicate keys K exist in the hash table. To address the issue of duplicate keys, after writing the pointer in

²The length unit can be changed as needed.

³The memory of the key-value block can be pre-allocated to reduce the latency of memory allocation in the critical path of insertion.

a bucket for an insertion, the client re-reads the two combined buckets to check duplicate keys, as shown in Figure 6b. On finding duplicate keys, the client only keeps one *valid* key and removes the remaining duplicate keys. Different clients have to determine the same key-value item as the valid key when finding duplicate keys in order to guarantee the consistency of a concurrent access. To guarantee this, we hence make an agreement in the algorithm to determine the only valid key for different clients. For example, within the two combined buckets, the agreement considers the key stored in the slot with the minimal bucket number and the minimal slot number to be the only valid one.

Lock-free Deletion. To delete a key-value item, the client first executes a search to find the target key. If the target key is found, the client sets its corresponding slot to be null by using an RDMA CAS, as shown in Figure 6c. Once the RDMA CAS is done successfully, the deletion request is returned. The client then sets the key-value block to full-zero and frees the key-value block in background. The zero-setting operation can be avoided if the RNIC can automatically set the freed memory to full-zero for data security, i.e., avoiding the old data to be observed by other clients.

Lock-free Update. To update a key-value item, the client searches the target key. At the same time, the client writes the new key-value item into the memory pool, as shown in Figure 6d. Once finding the target key exists, the client uses an RDMA CAS to change the content of the slot to point to the new key-value item. If the RDMA CAS is executed successfully, the update request is returned. The client finally frees the old key-value block in background.

Lock-free Search. As shown in Figure 6a, to search a key, the client reads its corresponding two combined buckets. If the fingerprint matches one of the slots, the client reads the key-value block that the slot points to. The client then compares the full key. If the full key matches, the value is returned.

Since all modifications on buckets are atomic and update requests do not modify the old key-value item in place, the only inconsistency case for a search is that the key-value block is freed or re-allocated before a search request reads the key-value block (after obtaining the pointer of the key-value). However, this inconsistency case can be easily observed by comparing the length and content of the key stored in the block with those of the search key. This is because once the key-value block is freed/re-allocated, its content is full-zero/changed, rendering the comparison mismatched. Nevertheless, there still exists a special case that another client re-allocates the key-value block and issues an RDMA WRITE to write the same key, key length, and value length as those of the old key-value block. As an RDMA WRITE is not atomic, it may write the key and key length completely but be writing the value. At this time, if reading the key-value block, a client can find the key is matched. But the value is broken, which cannot be observed by the client. To address this problem, we add a 64-bit checksum in each key-value block to enhance the

self-verification and check the integrity of a key-value block like Pilaf [31], as shown in Figure 5. Pilaf also shows that a 64-bit checksum is sufficient for verification.

Moreover, for insertion, deletion, and update requests, the operation of CASing a slot may fail, which means the slot is changed by another client before the CAS. In this case, RACE hashing re-searches the target key and then re-executes the failed insertion, deletion, or update request.

3.4 Extendible Remote Resizing

Using extendible resizing for disaggregated memory incurs two challenges, i.e., one extra remote access to read the directory for each index request and concurrent access during resizing, as presented in Section 2.3. In this subsection, we present a stale-read client directory cache scheme and a concurrent access scheme during resizing to address the two challenges respectively.

3.4.1 Client Directory Cache with Stale Reads

In order to reduce the extra RDMA READ for accessing the directory, we use a client directory cache for RACE hashing. However, caching the directory in clients incurs the data inconsistency issue between the directories in the memory pool and client caches. For example, when a client triggers a subtable resizing or directory resizing, the content of the directory in the memory pool is modified and thus the directories in the caches of other clients become stale. If other clients still query the hash table using their stale directories, they may locate an incorrect subtable and obtain incorrect data.

To address the inconsistency problem between client caches and the memory pool, in a *baseline solution* [17], and upon a client triggers a resizing operation, the client broadcasts a notification message to all other clients to invalidate their respective directory caches and does not start modifying the directory in the memory pool until receiving acks of all other clients. Obviously, the baseline solution incurs high performance overhead for resizing due to broadcasting messages and waiting for all acks. *The second solution* proposed by Pilaf [31] is to close the RDMA connections of all other clients to prevent these clients from performing RDMA READS once a resizing is triggered. Clients then re-connects the memory server to obtain the new table root after the resizing is completed. Pilaf addresses the problem of incorrect access but incurs high performance penalty due to blocking RDMA READS of clients. Therefore, both the solutions incur significant performance overheads.

In order to efficiently address this inconsistency problem, we propose a *stale-read client directory (SRCD) cache scheme* that does not need to broadcast messages or close the connections of other clients to the memory pool when triggering a resizing. Instead, by using the SRCD cache scheme, clients query the hash table still using the stale directories in their caches, but can verify whether the obtained data is correct. To achieve this, we add a header in each bucket of the RAC

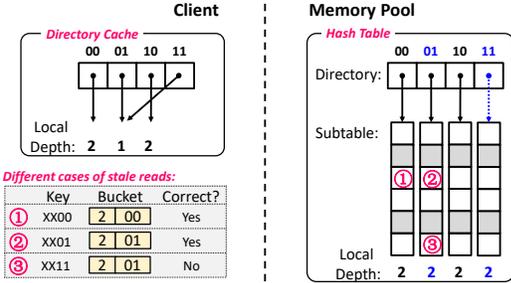


Figure 7: The stale-read client directory scheme. (Three cases when comparing the key with the fetched bucket header.)

hash subtable, as shown in Figure 5. The bucket header stores the local depth (LD) and suffix bits (Suffix) of the subtable that the bucket belongs to. The bucket header is not modified in IDU requests and is modified only when the subtable is created and resized. The local depth and suffix bits in the bucket header are used to verify whether the bucket is correct when executing search/insertion/deletion/update.

Figure 7 shows an illustration of using the SRCD cache scheme to verify the correctness of buckets. The client currently caches the directory of the hash table shown in Figure 2a, in which the directory entries “01” and “11” point to the same subtable. In the memory pool, the hash table is resized to be a new hash table shown in Figure 2b, in which a new subtable is created and the directory entries “01” and “11” point to different subtables. To search a key, the client first locates a subtable using the SRCD cache with stale reads and then fetches the buckets in this subtable via RDMA READs. After receiving a bucket, the client respectively compares the local depth and suffix bits stored in the bucket header with the local depth of the directory entry in the SRCD cache and the suffix bits of the key. The client can observe three cases as follows.

1) Both local depth and suffix bits match. If the local depth and suffix bits in the bucket header are respectively the same as the local depth of the directory entry in the cache and suffix bits of the key, the client can verify that the subtable is not resized and the fetched bucket is correct. For example, the key is “XX00” that corresponds to the directory entry “00”, i.e., Case ① in Figure 7.

2) Local depth mismatches and suffix bits matches. If the local depth in the bucket header is the same as that of the directory entry in the cache, the client knows the accessed subtable was resized in the memory pool. The client further computes the suffix bits of the key using the local depth stored in the bucket header and finds that the suffix bits of the key and those stored in the bucket header are matched. In this case, the client can verify the bucket is also correct although the subtable was resized. For example, the key is “XX01” but Subtable “01” was resized in the memory pool, i.e., Case ② in Figure 7. During the resizing, the keys with “11” are moved out Subtable “01” while the keys with “01” still remain in Subtable “01”. Therefore, when locating Subtable “01” to

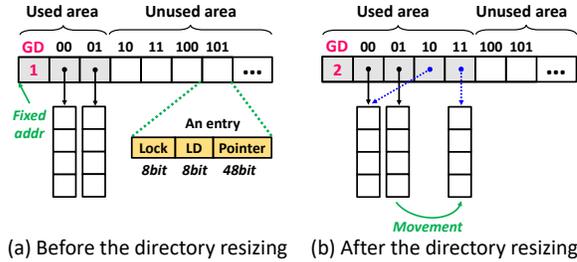


Figure 8: The resizing of the directory. (The global depth (GD) increases. The used area is not changed and new directory entries are written into the unused area.)

search the key with “01”, the client can obtain the correct key-value item.

3) Both local depth and suffix bits mismatch. If the local depth and suffix bits in the bucket header mismatch, the client can verify the subtable is resized and the searched key should be stored in the new subtable. For example, the key is “XX11” that corresponds to the directory entry “01” in the cache but Subtable “01” is resized in the memory pool and the key “XX11” is moved to the new Subtable “11”, i.e., Case ③ in Figure 7. In this case, the client fetches the new directory entries from the memory pool and re-executes the search.

In summary, only in Case “3) Both local depth and suffix bits mismatch”, the client needs to fetch new directory entries and update the local directory cache. In other cases, the client can locate correct buckets via stale reads.

3.4.2 Concurrent Access during Resizing

When an insertion failure occurs, a subtable resizing is triggered. During a resizing, we need to move slots from the resized subtable to the new one. Due to the slot movement, it is challenging to guarantee the correct execution of concurrent search, insertion, deletion, and update requests upon the subtable that is being resized. To address this challenge, we design the workflow of concurrent resizing as below.

To support concurrent access during resizing, the starting address of the directory in the memory pool cannot be changed. Otherwise, clients fail to find the new hash table after resizing. Therefore, we reserve a large enough contiguous memory space⁴ used for the future resizing of the directory. As shown in Figure 8, the directory includes a used area and an unused area. Clients only cache the used area. To resize the directory, e.g., increasing the GD from 1 to 2, the used area is not changed and the new directory entries are written into the unused area.

To resize a subtable, the client first locks the directory entry of the subtable in the memory pool. The lock only prevents other clients from resizing the same subtable but does not prevent other clients from executing search and IDU requests in the subtable. The client creates a new subtable and initializes the header of each bucket in the new subtable.

⁴For example, if we use at most 16 suffix bits for the directory, the memory space of 2^{16} directory entries is reserved.

The client further writes the pointer of the new subtable to the directory and locks the directory entry at the same time. The client then starts to move items. Figure 8b shows an example. The client moves items with Suffix “11” from Subtable “01” to Subtable “11”. The movement includes three steps for each bucket in Subtable “01”: ❶ updating the suffix bits in the bucket header from “1” to “2” (one RDMA CAS); ❷ inserting all items with Suffix “11” in this bucket into Subtable “11” (one or multiple RDMA CASes); ❸ deleting all items with Suffix “11” in this bucket (one or multiple RDMA CASes). By guaranteeing the order of executing the three steps, we support concurrent access to the subtable that is being resized. In the following, we discuss how to deal with the corner cases caused by the concurrent resizing below.

- *Concurrent search:* When executing a search, if finding that both local depth and suffix bits mismatch, the client can perceive the occurrence of the resizing in Subtable “01”. In this case, the movement may be before Step ❸ or after Step ❸. If the target key is found in the read bucket, it means the movement is before Step ❸. Otherwise, the movement is after Step ❸ or the key does not exist. The client further reads the bucket in Subtable “11” to lookup the target key.

- *Concurrent insertion:* During an insertion, the client re-reads the buckets to check duplicate keys, as shown in Figure 6b. To support concurrent insertion during resizing, we also check the bucket header after re-reading the bucket that the key is inserted to. If the bucket header is not changed, the insertion is successful. Otherwise, the client knows that a resizing occurs in the bucket. The client then compares suffix bits in the new bucket header with those of the inserted key. If the suffix bits match, the insertion is also successful. Otherwise, the client removes the pointer from Subtable “01” and re-inserts the key into Subtable “11”. Moreover, during a subtable resizing, an insertion may fail, i.e., not finding an empty slot in the subtable. In this case, the failed insertion triggers the next resizing. The next resizing will be blocked until the previous resizing releases the directory entry lock.

- *Concurrent deletion/update:* When executing a deletion/update, if finding that both local depth and suffix bits mismatch, the client waits for the completion of the movement and then deletes/updates the key from the new subtable. If the suffix bits match and Step ❶ occurs before the RDMA CAS operation of the deletion/update, there are two corner cases. First, if the RDMA CAS of the deletion/update fails, it means the item has been moved into the new subtable. The client will redo the deletion/update request in the new subtable. Second, the RDMA CAS of the deletion/update succeeds. But the client performing resizing operation fails to delete one item in Step ❸, which means that another client deleted/updated the item. The client performing resizing further cleans the pointer of the item in the new subtable and re-executes the item movement.

In summary, during a subtable resizing, all search/update/deletion and most insertion requests to the subtable are concur-

rently executed in a lock-free way. Only the failed insertions to the subtable are blocked due to triggering the next resizing.

4 Performance Evaluation

4.1 Experimental Setup

Testbed. We run all experiments on 5 machines, each with two 26-core Intel Xeon Gold 6278C CPUs, 384 GiB DRAM, and one 100Gbps Mellanox ConnectX-5 IB RNIC. Each RNIC is connected to a 100Gbps Mellanox IB switch. One machine is used for emulating the memory pool. To emulate the weak compute power, CPUs in the memory pool are only used for registering memory to RNICs during the initialization stage and do not involve in any requests of hashing indexes. The memory is registered with huge pages to reduce the page translation cache misses of RNICs [13]. The other machine is used for building the compute pool in which each CPU core serves as a client. Since current RNIC hardware does not support remote memory allocation in real time, we enable clients to pre-allocate all memory that future insertion and update requests require, which can also reduce the memory allocation latency in the critical path of request execution.

Workloads. We use YCSB [11] to evaluate the performance of different hashing indexes. We use the default Zipfian request distribution ($\theta = 0.99$) for all YCSB workloads. For most experiments, we use 16-byte keys and 32-byte values that are representative in real workloads of key-value stores [7, 15, 35]. We also evaluate the impact of different key-value sizes on the performance.

Comparisons. We compare RACE hashing with three state-of-the-art RDMA-search-friendly hashing indexes, i.e., Pilaf cuckoo hashing [31], FaRM hopscotch hashing [13], and DrTM cluster hashing [44]. Based on the optimal configurations presented in their papers, we use 3-way hashing and one slot per bucket in Pilaf cuckoo hashing, 4 slots per main bucket, 2 neighborhood, and 2 slots per overflow bucket in FaRM hopscotch hashing, and 8 slots per main or overflow bucket in DrTM cluster hashing. Moreover, since the disaggregated memory pool without CPUs cannot execute two-sided RDMA verbs, we implement these hashing indexes using only one-sided RDMA verbs as presented in Section 2.3 to facilitate a fair comparison. All key-value items are stored outside of the hash table to support variable-length keys and values. Each hash table is sized to store 100 million items.

4.2 Experimental Results and Analysis

4.2.1 Maximum Load Factor

The maximum load factor is defined as the ratio of the maximum number of stored items to the total number of slots in a hash table (including slots in main and overflow buckets), which is an important metric that affects the memory efficiency of a hash table. We insert unique keys into RACE, Pilaf cuckoo, FaRM hopscotch, DrTM cluster hash tables until an insertion failure occurs to evaluate their maximum load factors. Specifically, Pilaf cuckoo hashing reaches the maximum

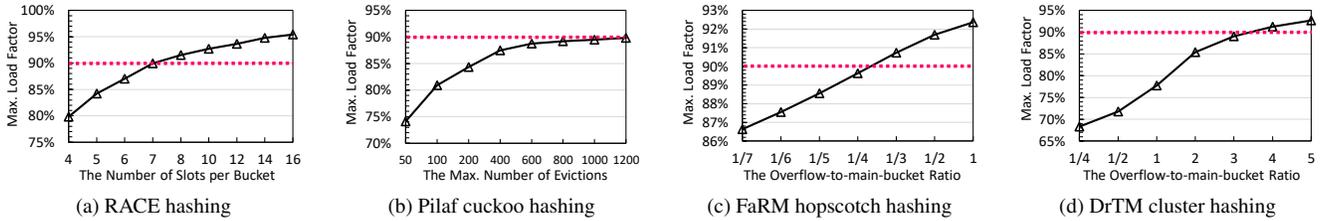


Figure 9: Maximum load factors of different hash tables. (In FaRM hopscotch hashing and DrTM cluster hashing, overflow buckets indicate the buckets linked in conflicting lists. The load factor indicates the ratio of the number of occupied slots to that of all slots in both overflow and main buckets. The overflow-to-main-bucket ratio means the ratio of the number of overflow buckets to that of main buckets.)

load factor when an insertion fails to lookup an empty slot after X cuckoo evictions. X means the maximum number of cuckoo evictions for an insertion and we evaluate maximum load factors of Pilaf cuckoo hashing under different X values. FaRM hopscotch hashing and DrTM cluster hashing reach their maximum load factors when running out of overflow buckets. As the ratio of the number of overflow buckets to that of main buckets (called overflow-to-main-bucket ratio) affects their maximum load factors, we evaluate their maximum load factors under different overflow-to-main-bucket ratios. For RACE hashing, since associativity (i.e., the number of slots per bucket) affects its maximum load factor, we evaluate its maximum load factors under different associativity.

As shown in Figure 9, the maximum load factor of RACE hashing increases with the increase of associativity. The maximum load factor of Pilaf cuckoo hashing increases with the increase of X . The maximum load factors of FaRM hopscotch hashing and DrTM cluster hashing increase with the increase of their overflow-to-main-bucket ratios.

To facilitate a fair comparison, we configure these hash tables to approach the same maximum load factor of 90% for the following experiments. RACE hashing reaches 90% when the associativity is 7. With 7 slots and one header, each bucket in RACE hashing is a cache-line size, i.e., 64B. Pilaf cuckoo hashing approaches 90% when $X = 1000$. FaRM hopscotch hashing and DrTM cluster hashing approach 90% when their overflow-to-main-bucket ratios are 1/4 and 3 respectively.

4.2.2 Execution Latency

To investigate request latencies of different hashing indexes, we respectively execute 1 million search, update, deletion, and insertion requests using one thread when these hash tables are in different load factors and evaluate average latencies of different requests, as shown in Figure 10. The items for search, update, and deletion are recently inserted [11].

Figure 10a shows the average insertion latencies of different hash tables. With the increase of load factors, the insertion latency of Pilaf cuckoo hashing exponentially increases due to causing more and more eviction operations and thus producing a large number of RDMA WRITES and locks; the insertion latency of FaRM hopscotch hashing dramatically increases due to linearly probing more buckets to find empty slots and

linking overflow buckets; the insertion latency of DrTM cluster hashing increases due to traversing longer bucket lists and linking new overflow buckets. The insertion latency of RACE hashing does not increase with the increase of load factors due to not causing any extra RDMA operations in which an insertion has 3 round-trip times (RTTs). When the hash tables are at the load factor of 90%, RACE hashing reduces the insertion latency by 1.9 \times , 8.8 \times , and 57.4 \times compared with DrTM cluster hashing, FaRM hopscotch hashing, and Pilaf cuckoo hashing respectively.

Figure 10b shows the average search latencies of different hash tables and all search requests are lock-free. A search in Pilaf cuckoo hashing needs 1.6 RTTs on average to serially read buckets and 1 RTT to read key-value block. A search in FaRM hopscotch hashing needs only 2 RTTs (one reads the neighborhood and the other reads the key-value block) at a low load factor and its latency increases in a high load factor due to traversing linked buckets. The search latency of DrTM cluster hashing sharply increases with the increase of the load factor since the bucket list becomes longer. When the hash tables are at the load factor of 90%, RACE hashing reduces the search latency by 2.3 \times , 1.2 \times , and 1.4 \times compared with DrTM cluster hashing, FaRM hopscotch hashing, and Pilaf cuckoo hashing respectively.

Figures 10c and 10d show the average deletion and update latencies of different hash tables, which deliver similar characteristics to those of search latencies. But deletion and update latencies of DrTM cluster hashing, FaRM hopscotch hashing, and Pilaf cuckoo hashing are much higher than their search latencies due to the needs of locking, unlocking, modifying slots, and unlinking buckets. The deletion and update latencies of RACE hashing are only higher than its search latency by 1-RTT latency. When the hash tables are at the load factor of 90%, RACE hashing reduces the deletion and update latencies by 1.8 – 2.3 \times and 1.6 – 2.2 \times respectively.

4.2.3 Concurrent Throughput

To investigate the concurrent request throughput of different hashing indexes, we first load 100 million items into a hash table and then successively execute 10 million searches, updates, deletions, and insertions to evaluate the concurrent throughput of different requests. We also vary the numbers of

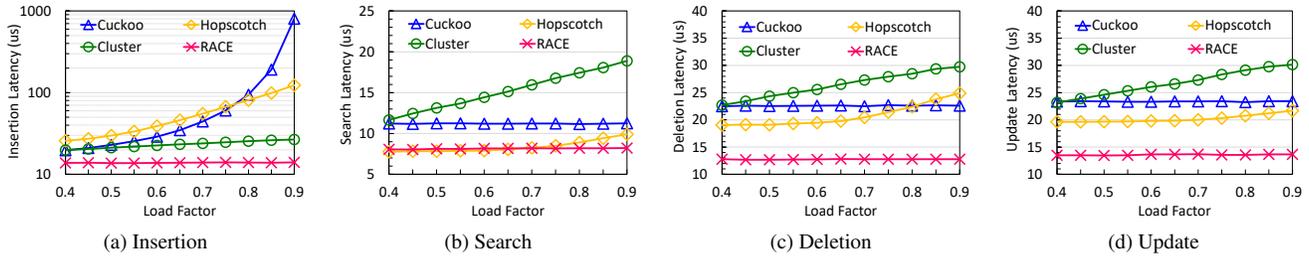


Figure 10: Average latencies of different requests when hash tables are in different load factors.

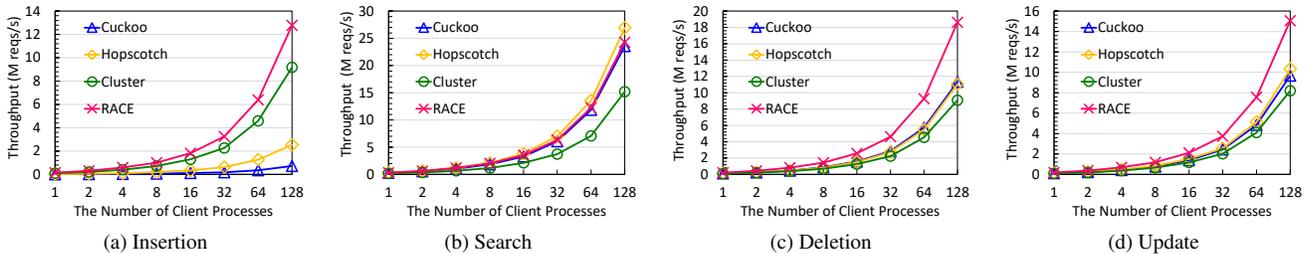


Figure 11: Concurrent throughput of different requests when using different numbers of client processes.

client processes to investigate the change of the throughput with the increase of clients, as shown in Figure 11. When the number of clients is not larger than 32, they are run in one client machine. When the number of clients is 64 and 128, they are run in 2 and 4 client machines respectively.

Figure 11a shows the concurrent throughput of insertion requests. The insertion throughputs of Pilaf cuckoo hashing and FaRM hopscotch hashing are much less than that of RACE hashing, of which reasons are the same as those of their high execution latencies. The insertion throughput of RACE hashing is 16.9 \times , 5.3 \times , and 1.4 \times on average higher than those of Pilaf cuckoo hashing, FaRM hopscotch hashing, and DrTM cluster hashing respectively.

Figure 11b shows the concurrent throughput of search requests. Pilaf cuckoo hashing, FaRM hopscotch hashing, and RACE hashing have a similar search throughput which is higher than that of DrTM cluster hashing. This is because DrTM cluster hashing needs to traverse linked bucket lists. The search throughput of RACE hashing is 1.7 \times on average higher than that of DrTM cluster hashing.

Figures 11c and 11d show the concurrent throughput of deletion and update requests respectively. The deletion and update throughput of RACE hashing is 1.7 – 2.1 \times and 1.5 – 1.9 \times higher than other hashing schemes due to the benefits of locking-free concurrency and RAC index structure.

4.2.4 YCSB Hybrid Workloads

To evaluate the throughput of different hashing indexes under YCSB hybrid workloads, we first load 90 million items into a hash table and then respectively run hybrid search/insertion workloads with different ratios. All tests use 128 client processes. The experimental results are shown in Figure 12. We observe that the throughput of all hashing indexes increases with the increase of search/insertion ratios, and RACE hashing

performs the best for all search/insertion ratios due to having both high search and insertion performance. Compared with DrTM cluster hashing, FaRM hopscotch hashing, and Pilaf cuckoo hashing, RACE hashing improves the performance of hybrid workloads by 1.4 \times , 4.9 \times , and 13.7 \times respectively when the search/insertion ratio is 10%/90%.

4.2.5 Variable-length Values

We increase the size of the key-value (KV) block from 64B to 8KB to evaluate the impact of variable-length KV sizes on the performance of RACE hashing, as shown in Figure 13. With the increase of the KV size, the latencies of insertion, deletion, update, and search requests increase due to reading and writing larger data. When the KV size is no larger than 512B, the increase of latencies is slight.

4.2.6 Extendible Remote Resizing

To support extendible remote resizing, we propose two techniques, i.e., the SRCD cache and concurrent access during resizing as presented in Section 3.4. We investigate the impact of the two techniques on the performance of RACE hashing.

Figure 14 shows the performance of RACE hashing with and without the SRCD cache. We observe that using the SRCD cache reduces 23%, 32%, 24%, and 23% of insertion, search, deletion, and update latencies respectively. This is because using the SRCD cache reduces one extra RDMA READ for accessing the directory.

To investigate concurrent access during resizing, we run two clients of which one executes insertions to trigger multiple resizings (the GD increases from 2 to 5) and the other executes random searches at the same time. We evaluate the average search latencies of RACE hashing with and without concurrent access during resizing as shown in Figure 15.

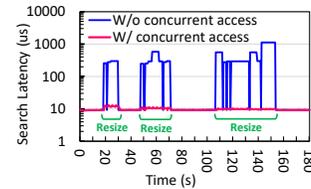
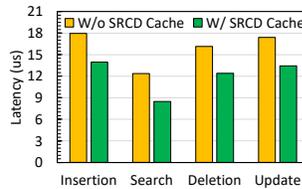
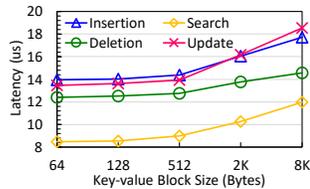
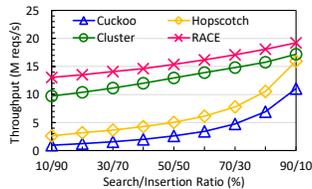


Figure 12: Hybrid workloads. Figure 13: Variable KV sizes.

Figure 14: The SRCD cache. Figure 15: Concurrent access.

Without the concurrent access, the average search latency during the resizing significantly increases since the searches stall until a resizing is completed. With the concurrent access, the average search latency during the resizing does not significantly increase and thus is about two orders lower than that without concurrent access.

5 Discussion

Concurrency Correctness. RACE hashing follows the concurrency correctness condition of *no lost keys* [30]: “a $get(K)$ operation must return a correct value for K , regardless of concurrent writers”. Specifically, when a search and an update run concurrently, the search can return either the new or the old value, while both of them should be unbroken and atomic. When a search and a deletion run concurrently, the search can return no key or the value that will be deleted.

Resizing Execution. In our current implementation, the client triggering the resizing itself performs the resizing. To improve the implementation, the client can create a background client/thread to perform the resizing.

Hardware Failure. Handling hardware failures including network failure, memory failure, and client CPU failure in the disaggregated memory architecture is complicated and tough. For example, after locking a directory entry, the client fails. To handle this failure, we need to enable other clients to perceive the failed client and release the lock directory entry or use the lease-based lock [44]. Our paper mainly focuses on the design of hashing index for disaggregated memory and plan to extend RACE hashing to support the handle of hardware failures in the future work.

6 Related Work

Memory Disaggregation. Memory disaggregation has recently received widespread attentions due to providing significant benefits for datacenters on resource utilization and scaling. Existing work studies various components in datacenters to support memory disaggregation including operating systems [38], hardware architectures [28, 29], memory managements [4, 37, 40–42], networks [1, 9, 12, 39], and new requirements [5, 18]. RACE hashing focuses on the design of index structures in the disaggregated memory which is orthogonal to these works.

Hashing Indexes on RDMA. With the wide use of RDMA in modern datacenters, RDMA-search-friendly hashing indexes have been intensively studied [13, 31, 44]. These hashing indexes are designed for traditional monolithic servers,

which however fails to efficiently work on the new disaggregated memory architecture, due to producing a large number of RDMA operations when executing IDU requests. RACE hashing is the first hashing index designed for disaggregated memory, in which all index requests can be efficiently executed by using only one-sided RDMA operations. Moreover, KV-direct [24] leverages programmable NICs with FPGA to offload hashing index operations, which is orthogonal to our paper that does not rely on FPGA.

Concurrent Hashing Indexes. Different concurrent hashing indexes are proposed to deliver high access throughput. MemC3 [15] uses a global lock to multi-reader and single-writer concurrency for concurrent cuckoo hashing. Libcukoo [26] leverages fine-grained locking to achieve multi-reader and multi-writer concurrent cuckoo hashing. Existing work [10, 33, 46] also proposes concurrent hashing indexes for persistent memory. However, all these existing schemes focus on concurrent access to local memory. Unlike them, our RACE hashing addresses the challenge of concurrent access to remote memory in hash indexes and enables all index requests to be executed in a lock-free manner.

7 Conclusion

This paper proposes RACE hashing, a one-sided RDMA-conscious extendible hashing index for disaggregated memory with lock-free remote concurrency control and efficient remote resizing. The hash table structure is designed to be one-sided RDMA-conscious, achieving that all index requests can be executed using only one-sided RDMA verbs while delivering high performance with constant-scale worst-case time complexity. Moreover, RACE hashing leverages a lock-free remote concurrency control scheme to enable index requests to be concurrently executed in a lock-free manner, and a stale-read client directory cache scheme to reduce one extra RDMA read for accessing the directory while guaranteeing the correctness of stale cache reads. We also achieve concurrent access to the subtable that is being resized. Experimental results show that RACE hashing outperforms state-of-the-art distributed in-memory hashing indexes by up to $13.7\times$ in YCSB hybrid workloads.

Acknowledgements

We sincerely thank our shepherd Roberto Palmieri and the anonymous reviewers for their insightful comments and suggestions.

References

- [1] Gen-Z technology. <https://genzconsortium.org/>.
- [2] Memcached - a distributed memory object caching system. <https://memcached.org/>.
- [3] Redis. <https://redis.io/>.
- [4] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC'18)*, pages 775–787, 2018.
- [5] Marcos K Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'19)*, pages 120–126, 2019.
- [6] Krste Asanović and David Patterson. FireBox: A hardware building block for 2020 warehouse-scale computers. In *Keynote of USENIX Conference on File and Storage Technologies (FAST'14)*, 2014.
- [7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems (SIGMETRICS'12)*, pages 53–64, 2012.
- [8] Mark S Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D Underwood, and Robert C Zak. Intel® omni-path architecture: Enabling scalable, high performance fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects (HOTI'15)*, pages 1–9. IEEE, 2015.
- [9] Amanda Carbonari and Ivan Beschastnikh. Tolerating faults in disaggregated datacenters. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets'17)*, pages 164–170, 2017.
- [10] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC'20)*, pages 799–812, 2020.
- [11] 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)*, pages 143–154, 2010.
- [12] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. R2C2: A network stack for rack-scale computers. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM'15)*, page 551–564, 2015.
- [13] 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, 2014.
- [14] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H Raymond Strong. Extendible hashing—a fast access method for dynamic files. *ACM Transactions on Database Systems (TODS)*, 4(3):315–344, 1979.
- [15] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, pages 371–384, 2013.
- [16] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojevic. Beyond processor-centric operating systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS'15)*, 2015.
- [17] Michael J Franklin, Michael J Carey, and Miron Livny. Transactional client-server cache consistency: Alternatives and performance. *ACM Transactions on Database Systems (TODS)*, 22(3):315–363, 1997.
- [18] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 249–264, 2016.
- [19] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *International Symposium on Distributed Computing (DISC'08)*, pages 350–364. Springer, 2008.
- [20] Hewlett Packard Corporation. The machine: A new kind of computer. <https://www.hpl.hp.com/research/systems-research/themachine/>.
- [21] Intel Corporation. Intel rack scale design architecture. <https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html>.
- [22] 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.

- [23] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*, page 468–479, 2013.
- [24] 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)*, page 137–152, 2017.
- [25] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*, page 476–488, 2015.
- [26] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems (Eurosys'14)*, pages 1–14, 2014.
- [27] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17)*, page 21–35, 2017.
- [28] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*, page 267–278, 2009.
- [29] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture (HPCA'12)*, pages 1–12. IEEE, 2012.
- [30] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (Eurosys'12)*, page 183–196, 2012.
- [31] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *2013 USENIX Annual Technical Conference (USENIX ATC'15)*, pages 103–114, 2013.
- [32] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.
- [33] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST'19)*, pages 31–44, 2019.
- [34] Nhan Nguyen and Philippas Tsigas. Lock-free cuckoo hashing. In *2014 IEEE 34th international conference on distributed computing systems (ICDCS'14)*, pages 627–636. IEEE, 2014.
- [35] 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 *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, pages 385–398, 2013.
- [36] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.
- [37] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 315–332, 2020.
- [38] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 69–87, 2018.
- [39] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A network architecture for disaggregated racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, pages 255–270, 2019.
- [40] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC'20)*, pages 33–48, 2020.
- [41] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, pages 306–324, 2017.
- [42] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru:

A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 261–280, 2020.

- [43] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE'18)*, pages 461–472. IEEE, 2018.
- [44] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, pages 87–104, 2015.
- [45] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proceedings of the VLDB Endowment*, 8(3):209–220, November 2014.
- [46] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 461–476, 2018.

Characterizing and Optimizing Remote Persistent Memory with RDMA and NVM

Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, Binyu Zang

Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Shanghai Artificial Intelligence Laboratory

Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

Abstract

The appealing properties of NVM including high performance, persistence, and byte-addressability, and a recent active thread of building remote memory systems with RDMA, have produced considerable interest in combining them for fast and persistent remote memory systems. However, many prior systems are either based on emulated NVM or have failed to fully exploit NVM characteristics, leading to sub-optimal performance.

This paper conducts a systematic study to summarize optimization hints that the system designer can use to exploit NVM with RDMA better. Specifically, we demonstrate how system configurations, NVM access patterns, and RDMA-aware optimizations affect the efficacy of RDMA-NVM systems. Based on the summarized hints, we empirically study the design of two existing RDMA-NVM systems, namely a distributed database (DrTM+H) and a distributed file system (Octopus). Both systems are designed when no production NVM is available, and neither of them achieves good performance on it. Our optimized systems achieve up to 2.4X (from 1.2X) better performance.

1 Introduction

People have been studying systems with emerging hardware technologies like Remote Direct Memory Access (RDMA) and Non-Volatile Memory (NVM) for many years, including but not limited to databases [8, 11, 25, 37, 39, 53], file systems [31, 32, 58], key-value stores [2, 10], and distributed shared memory systems [33, 38, 61]. These RDMA-NVM systems can either leverage NVM to persistently store the data or use it as DRAM to extend DRAM capacity.

Unfortunately, few systematic studies examined how to best leverage NVM with RDMA, since production NVM is only publicly available via Intel Optane DC persistent memory [36] (Optane PM¹) until recently. Except for a few systems [22, 33], prior work either uses emulated NVM [58, 61]

¹Since we exclusively study Optane PM in this paper, we use the terms NVM and Optane PM interchangeably throughout the paper.

or simply treats DRAM as NVM [8, 11, 32, 53]. Without such a study, system developers are unclear whether existing RDMA-NVM designs are efficient for Optane PM due to the following three reasons. First, emulating NVM with RDMA is particularly challenging because, to the best of our knowledge, most NVM emulators use CPU for the emulation [49]. However, RDMA may access NVM in a CPU-bypassing manner. Second, a recent study revealed that even the emulator could not faithfully simulate many Optane PM features [59]. Finally, a few systems evaluated with Optane PM do not consider its unique performance characteristics [33].

Our initial experiments further demonstrate that RDMA-NVM systems suffer from inferior performance when they treat NVM as DRAM. For example, the performance of *remote write* is far from the limits of NVM (§3) after switching the memory used in a *remote write* benchmark from DRAM to NVM: 16B one-sided RDMA WRITE only achieves 29% of NVM’s ideal write throughput. Hence, it is imperative to conduct a thorough study of the inferior performance and provide optimizations to mitigate the inefficiency.

There have been valuable studies on how to efficiently use NVM [59] with CPU and how CPU cache may affect the efficiency of RDMA with NVM [22]. Yang *et al.* [59] provide optimizations for the CPU to best utilize Optane PM. We study their findings on RDMA-NVM systems and confirm the importance of their optimizations. Nevertheless, we found some of the optimizations are suboptimal when considering RDMA. We further present optimizations that are best suited for RDMA on NVM. Kalia *et al.* [22] is the most relevant work: we share the same goal of improving RDMA’s performance with NVM. Specifically, they identify how CPU cache could hinder RDMA from fully utilizing NVM write bandwidth. We made a similar observation during our study. Besides, we also study other RDMA-NVM related factors, including inappropriate system configurations (§4.1) and application access patterns (§4.2).

Finally, existing systems use two network roundtrips to implement persistent write atop RDMA and NVM [20] because existing RDMA hardware is unaware of NVM. We ar-

gue that RDMA-NVM systems should consider broadly explored RDMA-aware optimizations [23, 24] to improve the persistent write performance with RDMA. With the help of known RDMA-aware optimizations, RDMA only needs one roundtrip for remote persistent write on the current hardware platforms (§4.3).

In this paper, we conduct a thorough systematic study on how to best utilize NVM with RDMA. Note that our focus is on *remote write*, i.e., the client issues write to the server NVM, either using one-sided or two-sided RDMA. The remote NVM read performance is close to that of DRAM (§3). To summarize, the contributions of this paper are:

A summary of optimization hints (H1–H9) to best utilize NVM with RDMA (§4). We study and collect various RDMA-NVM related optimizations—scattered from different sources—into one systematic study. We also propose new optimizations (H6–H8) that address the limitations of the existing study. The summarized hints are categorized into system configuration advice (§4.1), access pattern advice (§4.2) and RDMA-aware advice (§4.3). We empirically demonstrate how these hints help to fully utilize NVM for different RDMA primitives, i.e., RDMA can attain close to NVM write bandwidth and processing power.

An end-to-end study of improved RDMA-NVM system designs (§6). We use the summarized hints to analyze and improve the design of existing RDMA-NVM systems, namely a distributed database (DrTM+H [53]) and a distributed file system (Octopus [32]). We find that there is still significant room for improvement because both of them are designed when no production NVM is available. Our study helps to improve the throughput of DrTM+H by 1.5X and 2.2X for TPC-C [46] new-order transaction and Small-Bank [45], respectively. It also improves the I/O throughput of file data operations in Octopus by up to 2.4X (from 1.2X). These results strongly suggest we need further revisiting the design and implementations of existing RDMA-NVM systems, especially those not designed for Optane PM.

Our tools and benchmarks are available at <https://github.com/SJTU-IPADS/librdpma>.

2 Background

Figure 1 presents typical hardware components of a node with NVM in an RDMA-capable cluster. RDMA-capable NIC (RNIC) and NVM are attached to the processor, and they communicate with each other via the PCI Express (PCIe) bus.

2.1 Optane PM (NVM)

Intel Optane DC persistent memory [36] (Optane PM) is the first commercially available NVM. Besides a huge

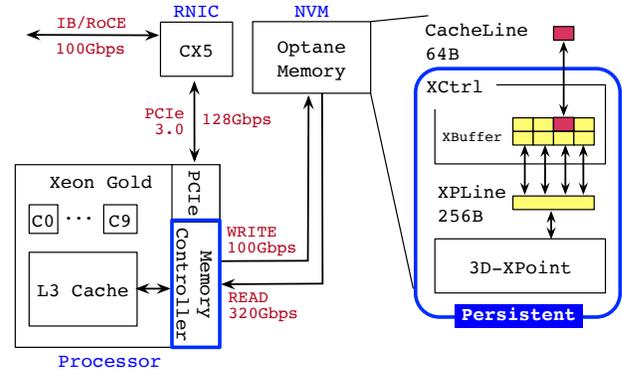


Figure 1. Hardware components of a node with NVM in an RDMA-capable cluster.

performance gain compared to traditional persistent storage (e.g., SSD), Optane PM also provides a DRAM-like memory interface. Thus, CPU can use load and store/non-temporal store² (ntstore) to read and write it, and RNIC can access it through PCIe read/write transactions.

Our study relies on an in-depth look at how Optane PM handles reads/writes. The right half of Figure 1 presents an overview of its components. Data is stored in NVM DIMMs (3D XPoint), while XController (XCtrl) transforms the read/write requests from the processor/PCIe into read/write requests to 3D XPoint. XCtrl has two important features. First, it receives requests in cacheline (CLine) granularity (64B), while 3D XPoint stores data in XPLine granularity (256B). Such a difference in granularity may incur read/write amplification. Second, in order to reduce write amplification, XCtrl has a small write-combining buffer (XBuffer) that merges adjacent cacheline writes into one XPLine write. Note that read requests also compete for XBuffer with write requests [59].

Persistent domain. Data is persistent once it reaches the node’s persistent domain. On the current hardware platform, the persistent domain comprises the Optane PM and the processor’s memory controller, as shown in Figure 1. Future hardware will further extend the persistent domain to the processor cache [4]. Nevertheless, the scope of the persistent domain is orthogonal to the results of this paper. We describe its impact in §5 in more detail.

Asymmetric performance feature. Optane PM is known for two asymmetric performance features. First, its read is much faster than write (320Gbps vs. 100Gbps). Second, sequential write has a higher bandwidth than random write due to the involvement of XBuffer.

Optane PM counters. Optane PM provides various useful counters³ that we use to analyze its behavior: NReadReq and

²non-temporal store has the same semantic as store except that it bypasses the CPU cache.
³Measured via ipmctl [5].

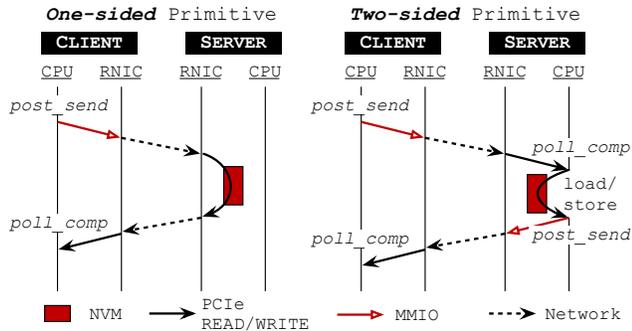


Figure 2. An overview of the interaction between RDMA and NVM.

`NWriteReq` record how many 64B read and write requests are received by XCtrl, while `NMediaRead` and `NMediaWrite` record how many bytes are read/written by 3D-XPoint Media. Based on these counters, we calculate the counter rates and use the counter rates to compute the read/write amplification of NVM: e.g., $\text{NMediaWrite rate} / (\text{NWriteReq rate} \times 64)$ measures the write amplification of Optane PM.

2.2 Remote direct memory access (RDMA)

RDMA is a fast networking feature with high throughput (e.g., 100Gbps bandwidth), low latency (e.g., $2\mu\text{s}$), and low CPU overhead. Representative implementations of RDMA include InfiniBand (IB) and RDMA over Converged Ethernet (RoCE). RDMA is well-known for its one-sided primitives (READ/WRITE⁴): RDMA-capable NIC (RNIC) can directly read/write the server memory bypassing the server CPU. RDMA also provides two-sided primitives (SEND/RECV) that are similar to message passing.

QP and the programming model of RDMA. RDMA hosts use queue pair (QP) to issue RDMA requests. The QP contains one *send queue* and one *completion queue*. To issue an RDMA request (e.g., one-sided RDMA READ), the host calls `post_send`, which uses memory-mapped IO (MMIO) to post the request to the *send queue*. If the host marks the request as *signaled*, then it can further obtain the completion event of the sent request, e.g., whether the payload of the READ has been fetched to the host, by polling the *completion queue* via `poll_comp`.

2.3 RDMA with NVM

Figure 2 shows how various RDMA primitives interact with Optane PM. One-sided RDMA primitives communicate with Optane PM through PCIe read/write transactions, while two-sided RDMA uses server CPU to read/write Optane PM⁵.

⁴We may use READ/WRITE as one-sided RDMA READ/WRITE.

⁵Although two-sided RDMA can use PCIe read/write transactions to write messages to Optane PM, we omit the discussion of such a case because its mechanism is the same as one-sided RDMA WRITE.

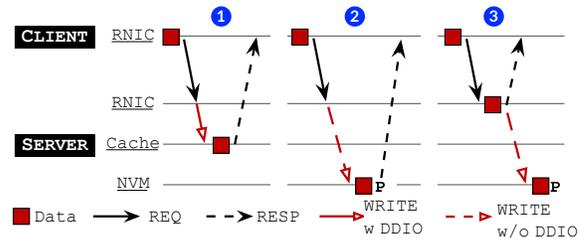


Figure 3. Three execution flows of using one-sided RDMA to write the server NVM. Note that without proper configurations (see §2.3), all three request flows are possible. The client uses the MMIO to post the WRITE request (REQ), and the client RNIC generates the response (RESP) via DMA. When DDIO (§2.3) is enabled, RNIC writes to the server’s last level cache (LLC). Otherwise, RNIC directly writes to Optane PM. **P** denotes the persistent point of the data, i.e., when the client can ensure the WRITE is persistent.

When different RDMA primitives access NVM, several factors may impact their efficacy:

Access granularity. RNIC, CPU and NVM (including XController and 3D XPoint) have different access granularities. Requests that do not match the device granularity cause extra read/write requests to the NVM. Hence, systems should carefully tune their NVM access patterns for different RDMA primitives (§4.2). Table 1 summarizes the access granularities of different devices.

Table 1: Access granularities of different hardware components.

	CPU	PCIe	XController	3D XPoint
Granularity	CLine	CLine	CLine	XPLine
Payload	64B	64B	64B	256B

DDIO. Data Direct I/O (DDIO) [16] aims to improve the server cache locality of the DMA-ed data, which allows the last level cache (i.e., L3 Cache) as the primary destination of the RNIC’s DMA-ed data (e.g., one-sided RDMA WRITE and two-sided RDMA). However, it is not friendly to Optane PM (§4.1).

Persistence. Two-sided primitives can use extended CPU instructions (e.g., `c1wb`) to ensure that the write to NVM is persistent. However, one-sided RDMA has no such instruction. Thus, one-sided RDMA-NVM WRITE is not persistent.

Figure 3 depicts three possible execution flows of one-sided RDMA-NVM WRITE on the current hardware platforms, only in the second case that the data is persistent (②). In the first case (①), the data is not persistent because it still resides in the volatile processor cache when the client receives the response. In the third case (③), the data is cached at the RNIC’s internal buffer when the client believes the write has finished. RNIC is not in the persistent domain (see Figure 1).

Table 2: *The configurations of machines in our testbed.*

Name	#	Hardware
Server	1	2x Intel Xeon Gold 5215M (10 cores), 384GB DRAM 2x ConnectX-5 IB RNIC (100Gbps) 1x 1.5T NVM (12x Optane DIMM)
Client	5	2x Intel Xeon E5-2650 v4 (12 cores), 128GB DRAM 2x ConnectX-4 IB RNIC (100Gbps)

Implementing persistent one-sided RDMA WRITE over current hardware (i.e., guarantee to achieve ② in Figure 3) requires specific configurations and extra one-sided requests: we should first disable DDIO to bypass the processor cache and then send an extra one-sided RDMA READ to the same QP issued the WRITE [19] to flush the previously cached WRITES. These two steps guarantee the WRITE is executed as the second case in Figure 3. However, a strawman implementation of this strategy uses two network roundtrips for a single write [20]. We describe optimizations for persistent WRITE in §4.3.

3 Testbed and Methodology

Testbed. Table 2 lists the hardware descriptions of our testbed. The server machine that equips Optane PM has two 10-core Intel Xeon Gold 5215M processors, 384GB DRAM and two ConnectX-5 MT27800 100Gbps Infiniband NIC. We attach six NVM DIMMs to each server’s processor, allowing them to achieve the maximum (ideal) bandwidth of Optane PM (320Gbps for read and 100Gbps for write). Each client machine has two 12-core Intel Xeon E5-2650 v4 processors, 128GB of DRAM, and two ConnectX-4 MCX455A 100Gbps Infiniband NIC. All machines are connected to a Mellanox SB7890 100Gbps InfiniBand Switch.

Target systems and evaluation methodology. The focus of this paper is on *remote write*, i.e., the client issues write requests to the server NVM using either one-sided or two-sided RDMA. For *remote read*, we find it has close performance to that of DRAM due to the asymmetric read/write performance feature (§2.1) of NVM.

To empirically analyze the read/write features of RDMA with NVM, we conduct a microbenchmark to evaluate the performance of *remote read* and *remote write* implemented by different RDMA primitives. In this benchmark, each client sends read/write requests with different payloads to the server’s NVM via RDMA, similar to prior work [10, 23, 40, 53]. The request addresses are chosen randomly. For one-sided RDMA, the client directly uses its primitives to implement *remote read* and *remote write*. For two-sided RDMA, the client sends messages to the server, and the server reads/writes NVM with `memcpy` after receiving the messages. We implement the benchmark on a state-of-the-art distributed execution framework designed for RDMA [53]. Unless other-

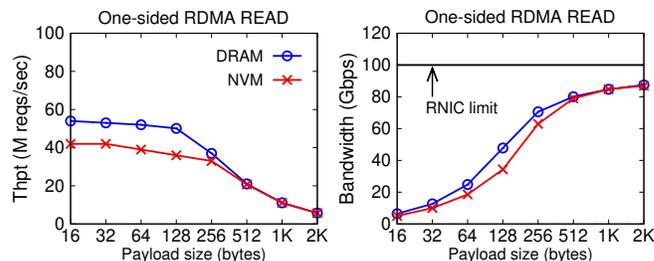


Figure 4. A comparison of one-sided RDMA READ performance on DRAM and NVM, (a) throughput and (b) aggregated bandwidth.

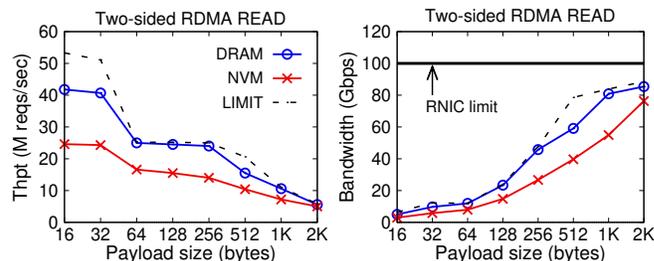


Figure 5. A comparison of two-sided RDMA READ performance on DRAM and NVM, (a) throughput and (b) aggregated bandwidth. LIMIT is measured when the server directly returns to the client without reading the payload.

wise mentioned, we report the per-socket peak throughput or bandwidth of reading/writing NVM through RDMA.

Figure 4 and Figure 5 present the performance of *remote read* on DRAM and NVM for one-sided and two-sided RDMA, respectively. For large payloads (e.g., 2,048B), the read performance of NVM is close to that of DRAM for both one-sided and two-sided primitives (99% and 90%). Reading NVM can hardly become a bottleneck in RDMA-NVM systems since NVM has a much higher read bandwidth than RNIC (320Gbps vs. 100Gbps). Note that for small reads (e.g., 16B), two-sided RDMA READ still suffers from obvious throughput degradation: it only achieves 59% of the DRAM read throughput. This is because CPU has much a higher read latency when reading NVM compared to DRAM (271ns vs. 82ns)⁶. In contrast, increased NVM read latency has negligible impact on one-sided RDMA READ since the PCIe latency is the dominant factor (~1000ns [35]).

Unlike *remote read*, the performance of *remote write* is much slower than that of DRAM, as shown in Figure 6 and Figure 7. More importantly, these results are not optimal because the measured performance is far from the theoretical limit of NVM or RDMA. For example, one-sided RDMA WRITE can achieve only 29% of the NVM peak write throughput (15M vs. 52M reqs/sec⁷) for small 16B writes.

⁶Measured by Intel Memory Latency Checker (MLC) [17].

⁷We estimate the NVM peak write throughput by dividing its peak write

Table 3: A summary of design advice, optimization hints, and whether the hints can apply to a specific RDMA primitive. ✓ indicates a positive optimization effect, and “-” means the hint does not target the case.

Design advice	Optimization hints	One-sided	Two-sided
A1. Configuration (§4.1)	H1. Avoid cross-socket NVM accesses	✓	✓
	H2. Limit concurrent access to a single NVM DIMM for two-sided RDMA	-	✓
	H3. Disable DDIO; if DDIO must be enabled, use two-sided RDMA	✓	-
A2. Access pattern (§4.2)	H4. Use <code>ntstore</code> instead of <code>store</code> for large writes	-	✓
	H5. Use XPLine granularity for writes	✓	✓
	H6. Use PCIe DW granularity (64B) for small writes (i.e., less than XPLine)	✓	-
	H7. Use cacheline granularity (64B) with <code>ntstore</code> for small writes (i.e., less than XPLine)	-	✓
	H8. Use less atomic operations on NVM	✓	✓
A3. RDMA-aware (§4.3)	H9. Enable outstanding request with doorbell batching for one-sided persistent WRITE	✓	-

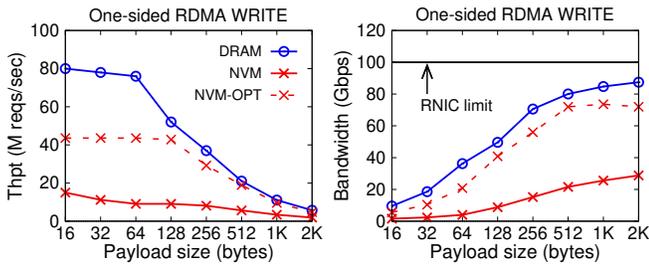


Figure 6. A comparison of one-sided RDMA WRITE performance on DRAM and NVM, (a) throughput and (b) aggregated bandwidth. NVM-OPT applies the optimizations from §4.

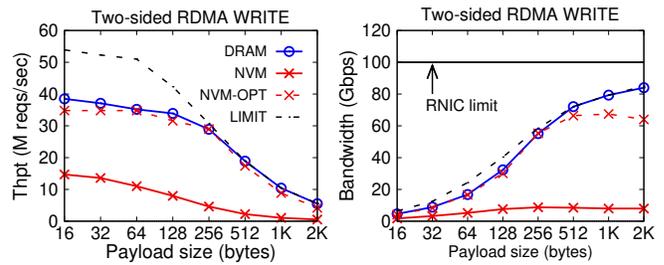


Figure 7. A comparison of two-sided RDMA WRITE performance on DRAM and NVM, (a) throughput and (b) aggregated bandwidth. LIMIT is measured when the server directly returns to the client without writing the payload. NVM-OPT applies the optimizations from §4.

Further, one-sided and two-sided RDMA saturate only 38% and 12.5% of the NVM’s peak write bandwidth for large 2,048B writes, respectively. Therefore, there is significant room for improvement.

Our approach. To understand why *remote write* has inferior performance, we conduct a systematic study to summarize various performance-relevant factors for RDMA to write NVM. Inspired by a recent CPU-specific NVM study [59], we mainly follow two directions. First, we investigate what system configurations can affect RDMA’s efficiency with NVM (§4.1). Second, we study which access patterns from RDMA are friendly to NVM (§4.2). For each direction, we empirically study whether known NVM optimizations are necessary or optimal for RDMA. The results show that some setups are not necessary, while some optimizations are sub-optimal for RDMA. To this end, we present new optimizations by fully considering NVM characteristics with RDMA. Finally, we use existing RDMA optimizations to improve the persistent write of one-sided RDMA atop of NVM (§4.3).

bandwidth (100Gbps) with the size of XPLine (256B).

A preview of the optimization results. Figure 6 and Figure 7 present our optimized version of one-sided and two-sided RDMA NVM write (*NVM-opt*). After applying all the optimizations, they have significantly better performance and achieve close to the NVM limit. For example, one-sided 16B RDMA NVM WRITE achieves 45M reqs/sec, 87% of the ideal peak throughput of NVM write.

4 Design Advice

This section summarizes design advice and optimization hints for high-performance RDMA-NVM systems, as shown in Table 3. We present both new optimizations (e.g., Hint 6, **H6**) and brief descriptions of known optimizations. Further, we show that some known optimizations that only consider CPU accessing NVM are sub-optimal for RDMA (e.g., **H5**). Among these optimizations, **H1–H8** applies to systems that use Optane PM as volatile storage and persistent storage, while **H9** only targets systems that use Optane PM as per-

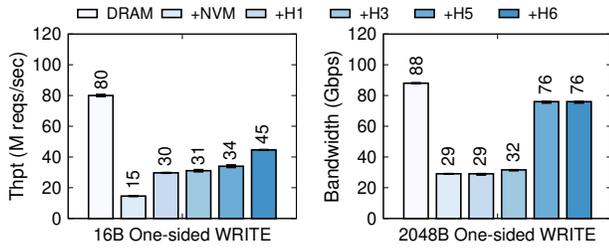


Figure 8. Factor analysis of the optimizations used to improve NVM write performance atop of one-sided RDMA WRITE under (a) small payload (16B) and (b) large payload (2,048B). We do not include **H8** as it does not target remote write. Note that the error bars are small.

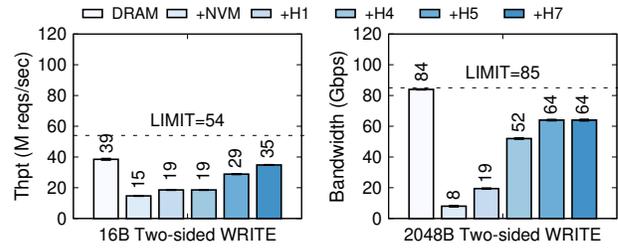


Figure 9. Factor analysis of the optimizations used to improve NVM write performance atop of two-sided RDMA for (a) small payload (16B) and (b) large payload (2,048B). LIMIT is measured as the server directly returns to the client without writing the payload. We do not include **H8** as it does not target remote write. Note that the error bars are small.

sistent storage.

We make two assumptions about the hardware components. First, the RNIC is a PCIe-based device, which holds for most existing RNICs [24]. Second, the NVM is Optane PM [36], the first (and only) commercially available NVM device. Unless otherwise stated, we use the same *remote write* microbenchmark in §3 for the study.

4.1 Configuration advice

A prior CPU-specific NVM study [59] has provided valuable configuration setups (e.g., NUMA setup) for the CPU to better utilize NVM. We summarize these setups in **H1** and **H2**. A natural question to answer is: do RDMA-NVM systems require the same setups? Our study reveals that first, RDMA-NVM systems should also consider **H1**. Meanwhile, one-sided RDMA does not necessarily require **H2** as the CPU. Finally, RDMA introduces a new configuration option, **H3**. Succinctly, the configuration advice for RDMA-NVM systems is the following three optimization hints:

- H1.** Avoid cross-socket NVM accesses;
- H2.** Limit concurrent access to a single NVM DIMM for two-sided RDMA;
- H3.** Disable DDIO; If DDIO must be enabled, use two-sided RDMA for large NVM writes;

Hint H1. Yang *et al.* [59] found that the NVM write bandwidth of a socket could be halved from other sockets. Since an RNIC leverages its attached socket to access NVM from another socket (see Figure 1), slow cross-socket NVM accesses also impact RDMA-NVM systems. Figure 9 and Figure 8 illustrate this: removing cross-socket access improves the baseline two-sided and one-sided RDMA write performance by up to 2.4X (8Gbps vs. 19Gbps) and 2X (15M vs. 30M reqs/sec), respectively. Thus, RDMA-NVM systems should also avoid cross-socket NVM accesses.

Apply H1. Readers may wonder whether **H1** is feasible in real systems. One strategy to apply **H1** is first attaching one RNIC for each socket, and then treating each socket as a logical node in a cluster. Such a setup is common in RDMA-capable systems [11, 29, 51, 54, 62], and it naturally avoids cross-socket NVM access. Furthermore, even if there are insufficient RNICs for each socket to have a dedicated RNIC, one can adopt the techniques in IOctopus [42] to apply **H1**. Using IOctopus, one RNIC can simultaneously communicate with multiple sockets. Hence, RDMA can directly access the NVM bypassing the attached socket.

Hint H2. Another observation from Yang *et al.* [59] is that the CPU fails to scale up when writing to a single NVM DIMM. This phenomenon also affects two-sided RDMA because it uses CPU to write to the NVM. As shown in Figure 10(a), compared to using four threads at the server to handle writes, two-sided RDMA-NVM write bandwidth drops 37% when using 20 threads. Note that to avoid interference from other factors, we have enabled all other optimizations hints in this experiment. Therefore, system designers should also reduce concurrent access to a single NVM DIMM for two-sided RDMA. In practice, designers can control which DIMM to access by selecting the appropriate NVM addresses [59]. For example, assuming the starting address of the NVM is 4KB-aligned, and the NVM is interleaved on 6 DIMMS, the first and seventh 4KB is on the first DIMM, and the second 4KB is on the second DIMM, etc.

Does one-sided RDMA suffer from the same issue? To quantify this, we further conduct an experiment to measure the concurrent write performance of one-sided RDMA-NVM WRITE. In this benchmark, we increase the number of clients that send concurrent one-sided RDMA WRITE to a single NVM DIMM, and measure their aggregated bandwidth. Figure 10(b) presents the results: one-sided RDMA WRITE scales well with the increased number of concurrent requests. This implies that one-sided RDMA is more robust when concurrently accessing a single NVM DIMM.

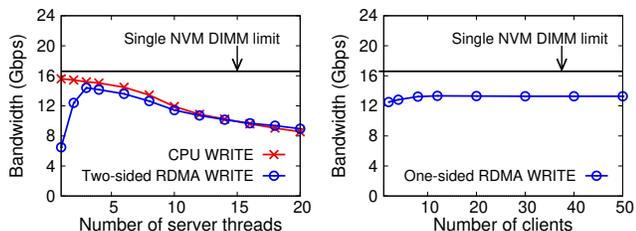


Figure 10. Effects of concurrent accesses to a single NVM DIMM for (a) two-sided RDMA and (b) one-sided RDMA WRITE using a 2048B payload. The write bandwidth limit of a single NVM DIMM is about 16Gbps (one-sixth of the evaluating Optane PM). Note that we have enabled all other optimizations for both RDMA primitives in this experiment.

To the best of our knowledge, it remains unknown why the CPU cannot scale up when accessing a single NVM DIMM. Yang *et al.* [59] suspected that the high NVM access latency may cause head-of-line blocking effects at the processor. Similarly, we suspect that the RNIC can scale well for one-sided RDMA because the increased latency of NVM access is not that significant compared to PCIe latency.

Hint H3. One important RDMA-specific configuration is whether to enable DDIO, which controls the destination of one-sided RDMA WRITE (§2.3). A prior study has revealed that DDIO has a huge performance impact on one-sided RDMA-NVM WRITE for large payloads [22]; we also made a similar observation during our study. Consequently, **H3** is an important configuration setup for RDMA-NVM systems.

Impact of DDIO. Figure 11(a) shows the effects of DDIO on one-sided RDMA-NVM WRITE. Since large RDMA-NVM WRITE is more sensitive to DDIO, we use a bulk write benchmark where a single client issues a sufficient large payload to measure the peak bandwidth of WRITE. With DDIO enabled, we observe that WRITE can only reach half of the NVM peak bandwidth (42Gbps vs. 100Gbps). On the other hand, the WRITE can achieve close to NVM peak bandwidth with DDIO disabled.

Ideally, the client should saturate the RNIC(NVM) bandwidth in this benchmark. However, it fails because DDIO changes the sequential writes from RNIC to random writes to NVM. Figure 12(a) illustrates this: the RNIC first sequentially writes the data into the cache, after that the cache randomly evicts the data to the NVM. Random writes cannot saturate NVM bandwidth because NVM has less chance to merge adjacent writes to avoid write amplification (see §2.1).

Measuring the write amplification of DDIO. To quantify the effect of DDIO, Figure 11(b) analyzes the write amplification of NVM⁸ with different configurations. We can see that

⁸We measure the write amplification of DDIO via NVM counters. §2.1 describes the measurements in more detail.

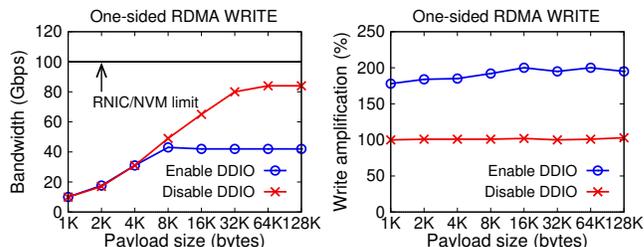


Figure 11. (a) Effects of DDIO to bulk one-sided RDMA-NVM WRITE, (b) write amplification analysis of one-sided RDMA-NVM WRITE. Note that we have enabled all other optimizations in this experiment.

enabling DDIO incurs a roughly 2X write amplification to NVM WRITE, which explains why the bandwidth is halved for a large payload (e.g., more than 64KB).

Implications for RDMA-NVM systems. If the systems must use one-sided RDMA WRITE to saturate the NVM bandwidth, we recommend considering **H3** to turn off the DDIO first. The system can statically enable/disable DDIO via the BIOS setups [58] or dynamically adjust the bits in Integrated I/O (IIO) Configuration Registers [1, 13, 18] at runtime. We follow prior work [13] to use configuration registers to configure DDIO at runtime.

Limitations of disabling DDIO. On the current hardware platform, the DDIO configuration affects all the devices on a processor. Thus, server CPU will have a poorer cache locality for DMA-ed data (e.g., messages in two-sided primitives) with DDIO disabled, resulting in degraded two-sided RDMA performance. For example, we measured a 57% peak throughput drop (54M vs. 23M reqs/sec) of two-sided RDMA primitives after disabling DDIO. Therefore, the current RDMA-NVM systems will make a trade-off between the bandwidth of one-sided RDMA NVM WRITE and the performance of two-sided RDMA. Considering the limitations of disabling DDIO, we extend **H3** as follows:

H3 (extended). If DDIO must be enabled, use two-sided RDMA for large NVM writes;

Two-sided RDMA can leverage the CPU at the server to fully utilize NVM [59]. Hence, RDMA-NVM systems can adopt a hybrid approach for implementing NVM write based on the request payload size.

4.2 Access pattern advice

Tuning systems NVM access pattern according to the Optane PM's features is critical to NVM-aware systems. Known optimizations for CPU include choosing the appropriate CPU instructions and using the proper access granularity [59]. We summarize these hints in **H4–H5**:

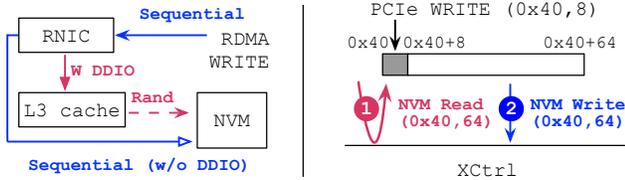


Figure 12. (a) DDIO changes the sequential accesses of RNIC into random accesses. (b) An example of PCIe partial write: PCIe sends two NVM requests when writing 8B at address 0x40.

H4. Use `ntstore` instead of `store` for large writes;

H5. Use XPLine granularity for writes;

Both hints can also benefit RDMA-NVM systems. However, we found **H5** is not optimal when considering RDMA, especially for small writes, because it incurs huge network amplification. For example, applying **H5** only improves the performance of 16B one-sided RDMA NVM WRITE by 1.1X (31M vs. 34M reqs/sec), which remains far from NVM’s ideal processing rate (52M reqs/sec). In this case, **H5** will incur 16X (256B vs. 16B) network amplification to one-sided RDMA WRITE. Since moving 256B data to DRAM over RDMA is even slower than NVM ideal processing rate (37M vs. 52M reqs/sec, as shown in Figure 6), the network would first become the bottleneck for small writes.

To this end, we found the key for small writes to fully utilize NVM is to *avoid sending unnecessary read requests to the NVM*. As we have mentioned in §2.1, NVM read requests compete for XCtrl processing power with NVM write requests. Hence, unnecessary read requests would drastically reduce the NVM write throughput. This fact allows using a smaller access granularity to saturate NVM’s processing rate with RDMA.

CPU and RNIC generate an extra read request to NVM if the payload of the write request does not fit their access granularities (i.e., cacheline and PCIe DW). They execute such a write request in a read-modify-write pattern to avoid overwriting the original content. Thus, using the CPU/RNIC access granularity is sufficient to prevent sending unnecessary reads from the device to NVM. Based on this observation, we first propose **H6–H7** to complement **H5** for small writes. Further, since the read-modify-write pattern is not friendly to NVM, we also propose **H8** to suggest systems use fewer such operations. Specifically, we propose the following hints to complement **H4–H5**:

H6. For one-sided RDMA, use PCIe data word (64B) granularity when the payload is smaller than XPLine;

H7. For two-sided RDMA, use cacheline granularity (64B) with `ntstore` when the payload is smaller than XPLine;

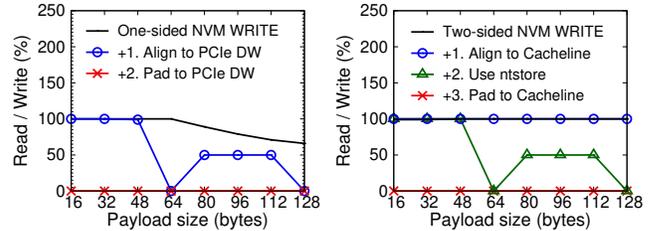


Figure 13. The ratio of extra NVM read per write of (a) one-sided RDMA WRITE and (b) two-sided RDMA on the remote write benchmark. Aligning and padding the write payload to device access granularity (e.g., PCIe DW) is sufficient to avoid unnecessary NVM reads for one-sided RDMA WRITE. On the other hand, two-sided RDMA further needs to use `ntstore`.

H8. Use less atomic operations on NVM;

Hint H6. PCIe issues write in a read-modify-write pattern with PCIe partial-write. Figure 12(b) shows a concrete example where the RNIC uses PCIe to write 8B at 0x40. It will first send a read request to the NVM to read the data word at 0x40 (1). Then, it overwrites the entire data word according to the write request (2).

Figure 8(a) presents the optimized performance of **H6** to one-sided RDMA WRITE: it improves the performance of 16B WRITE to 87% of the NVM’s peak write throughput (45M vs. 52M reqs/sec). The result is 1.3X faster than applying **H5** thanks to the reduced network amplification. Figure 13(a) further examines how eliminating PCIe partial write helps to prevent sending read requests to the NVM. We apply **H6** by first aligning the written address to PCIe DW (+1. Align to PCIe DW), and then padding the payload size (+2. Pad to PCIe DW) to a multiple of PCIe DW. As we can see, after applying both steps, one-sided RDMA WRITE does not issue a read request to the NVM. Consequently, small WRITES (e.g., no larger than 64B) can reach close to the NVM processing limit.

We should mention that reducing the PCIe partial write may also benefit one-sided RDMA-DRAM WRITE. However, our experiments show that it may even have a negative effect on DRAM WRITE. For example, the 16B DRAM WRITE has a 25% performance degradation on our testbed after applying **H6**.

Hint H7. Similar with **H6**, **H7** suggests how to prevent read-modify-write for two-sided RDMA. As shown in Figure 8, it improves the 16B two-sided RDMA-NVM WRITE performance by 1.8X (19M vs. 35M reqs/sec). To apply **H7**, two-sided RDMA should use `ntstore` together with use cacheline granularity (+1. Align to Cacheline and +3. Pad to Cacheline, as shown in Figure 13(b)). This is because, the CPU would pre-fetch the cacheline using NVM read with

store. As shown in Figure 13(b), two-sided RDMA-NVM WRITE always achieves a 100% read/write ratio even after using the proper access granularity.

Hint H8. A lesson learned from H7–H8 is that the read-modify-write access pattern is not friendly to NVM. Atomic operations (e.g., one-sided RDMA ATOMIC compare and swap) naturally follow a read-modify-write pattern. Worse, the designer cannot apply prior hints to optimize atomic operations. For instance, one-sided RDMA ATOMICs cannot apply H6 since they use a fixed 8B granularity. Consequently, we suggest using fewer atomics on NVM for RDMA. Note that we are not suggesting disabling atomics, but *moving the data for atomic operations (e.g., spinlock) from NVM to DRAM whenever possible*.

Discussion of H6–H8. Although applying H6 and H7 may waste NVM storage for storing the padding, while adopting H8 could change the persistent semantic of atomic data, we believe H6–H8 is actionable in real systems. This is because there are many scenarios in RDMA-NVM systems that can use H6–H8 without wasting storage or changing the application semantics. For instance, existing RDMA-NVM enabled databases [10, 11, 25, 53, 54] do not require the lock to be persistent. Thus, we can safely move their lock metadata from NVM to DRAM. Furthermore, distributed logging [10, 11] naturally uses padding to accommodate future logs. Thus, applying H6 to logging does not introduce additional storage overhead. Finally, as we will present in §6, H6–H8 can have huge performance improvements for existing systems. For example, H6–H8 can improve the performance of DrTM+H on the SmallBank [45] benchmark by 1.79X (3.9M vs. 7.0M txns/sec, see Figure 16).

4.3 RDMA-aware advice

Finally, we discuss how known RDMA-aware optimizations can mitigate the inefficiency of implementing persistent write atop of existing hardware platforms. As we have mentioned in the introduction, a strawman approach to implementing persistent write using one-sided RDMA requires two network roundtrips: the first WRITE attempts to store the data to NVM, while the second READ ensures that the written data is flushed to the persistent domain (e.g., Optane PM). Fortunately, it is possible to leverage well-known RDMA-aware optimizations to avoid the additional network roundtrip of READ. H9 summarizes this fact:

H9. Enable outstanding request with doorbell batching for one-sided persistent RDMA WRITE.

Specifically, outstanding request [23] allows us using the completion of READ as the completion of the WRITE, as long as the two requests are sent to the same QP. Since

the READ to persist the WRITE must be post to the same QP as the WRITE (§2.3), we no longer need to wait for the first WRITE to complete. Thus, this optimization reduces the wait time of the first network roundtrip. Applying outstanding request to persistent WRITE is correct because first, later READ flushes previously WRITE [19], and RNIC processes requests from the same QP in a FIFO order [6].

Based on outstanding request, doorbell batching [24] further allows us to send the READ and WRITE in one request using the more CPU and bandwidth efficient DMA, reducing the latency of posting RDMA requests.

On our testbed, a single one-sided RDMA request takes $2\mu\text{s}$. Thus, a strawman implementation of *remote persistent write* uses $4\mu\text{s}$. After applying H9, one-sided *remote persistent write* takes $3\mu\text{s}$ latency to finish.

5 Discussion of Future Trends

Generality of the study. Our study focuses on specific RNIC (Mellanox ConnectX-5) and NVM (Intel Optane DC Persistent Memory), while other hardware devices may yield different results. Nevertheless, we believe ConnectX-5 is a representative RNIC, as recent generations of Mellanox RNICs (e.g., Connect-IB, ConnectX-4) all share the same architecture. Moreover, Optane PM is the only commercially available NVM. Finally, we also provide open-source tools that the developers can use to examine their design choices under different hardware configurations.

Next-generation NVM. The next-generation of NVM not only has a better performance but also provides a larger scope of persistent domain. First, it will have a 25% higher bandwidth [21]. Second, it will include the processor cache in its persistent domain [4]. This feature is desirable for one-sided RDMA since one-sided RDMA no longer depends on DDIO for WRITE to be persistent (§2.3). Consequently, the designer does not need to make a trade-off between one-sided persistence and two-sided RDMA performance (§4.1).

On the other hand, the new features of next-generation NVM are unlikely to change the advice of our study. First, the primary focus of our study is *how to avoid NVM write becoming the bottleneck* in RDMA-NVM systems (§3), even when NVM write has a comparable performance with RDMA (see Figure 1). Thus, the 25% bandwidth improvement of the next-generation NVM is insufficient to twist the performance comparisons between RDMA and NVM, since future generations of RDMA will have much higher bandwidth. For example, RNIC with 200Gbps bandwidth has already been commercially available [3], which is 2X higher than our evaluated RNIC. Besides performance, the enhanced functionality of next-generation NVM, i.e., putting cache in the persistent domain, is also unlikely to address the current performance issue caused by the cache. This is because the random cache eviction is still not suitable for

NVM. Meanwhile, an extra one-sided RDMA READ is still required to ensure persistence as long as the RNIC is not redesigned.

Suggestions to hardware designers. There are proposals to extend RDMA to cooperate with NVM, e.g., Talpey *et al.* [44] proposed to add a *one-sided commit* primitive to support one-side persistent RDMA WRITE. Nevertheless, our study reveals that existing proposals are insufficient: *the hardware designers not only need to consider hardware extensions to support more functionality, but also should consider extensions for better performance.* For instance, adding an RDMA-version of *ntstore*, e.g., one-sided *non-temporal* RDMA WRITE that allows the WRITE to bypass the cache—can greatly improve the flexibility in configuring DDIO for RDMA-NVM systems (§4.1).

6 Improved System Designs

Existing or future RDMA-NVM systems can benefit from the summarized optimization hints in our study (§4). In this section, we present how we use these hints to improve the performance of two open-source RDMA-NVM systems, a distributed database (DrTM+H [53]), and a distributed file system (Octopus [32]). Both systems are designed when no production NVM is available.

6.1 Distributed database

DrTM+H [53] is a distributed transactional system designed for RDMA and NVM. It fully leverages the power of one-sided and two-sided RDMA to boost transaction execution. We choose it for optimization for two reasons. First, its concurrency control and replication protocol use RDMA and NVM heavily. Therefore, there may exist a huge space for improvements. Second, most existing RDMA-NVM distributed databases adopt a similar protocol [8, 10, 11, 25, 37] as DrTM+H. Thus, our improved DrTM+H design can potentially benefit these systems.

Overview. DrTM+H uses optimistic concurrency control [28] (OCC) to ensure strict serializability and primary-backup replication to achieve high availability [8, 11]. It organizes NVM as a distributed shared memory pool similar to prior work [10, 32, 54], which stores the database records and transaction logs. DrTM+H uses four phases to execute a transaction, each interacts with NVM using RDMA as follows: *Execution* uses one-sided RDMA READ to read records stored in NVM; *Validation* uses one-sided RDMA Compare and swap (CAS) to acquire the lock co-located with the record; *Logging* uses one-sided RDMA WRITE to replicate the transaction updates to backups; *Commit* uses two-sided RDMA to update and unlock the records.

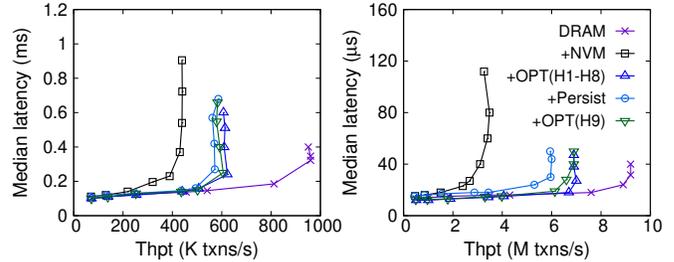


Figure 14. The performance of DrTM+H on (a) TPC-C/no and (b) SmallBank.

6.1.1 Optimizations

The original DrTM+H neither considers the performance features of NVM (§2.1), nor the persistence issue of one-sided RDMA WRITE (§2.3). In this section, we first apply optimizations that are beneficial to DrTM+H both when using NVM to extend DRAM capacity and to support durability (i.e., **H1–H8**):

- Separate the memory pool from different sockets to avoid cross-socket NVM access (**H1**).
- Configure DrTM+H with DDIO disabled (**H3**).
- Use *ntstore* to optimize the *commit phase* (**H4**).
- Align and pad logs/records larger than 256B to XPLine granularity (**H5**).
- Align and pad logs/records smaller than 256B to 64B granularity (**H6 + H7**).
- Implement a DRAM-based lock service for the *validation phase* (**H8**). Note that it is safe not persisting the locks in NVM because DrTM+H does not require the locks to be persistent even when durability is enabled.

Second, we use **H9** to optimize DrTM+H when use NVM to support durability. When following existing approach [20] to support durable transactions, DrTM+H has to use two network roundtrips to persist a single transaction log at the logging phase. With the help of **H9**, the logging phase only use one roundtrip:

- Implement remote persistent log with **H9** in one roundtrip.

6.1.2 Evaluation

Setup. By default, DrTM+H executes transactions in a symmetric setting [10]: each machine both stores database partition and executes transactions. Nevertheless, we evaluate it in an asymmetric setting due to the lack of NVM-capable machines (Table 2): the NVM server stores the

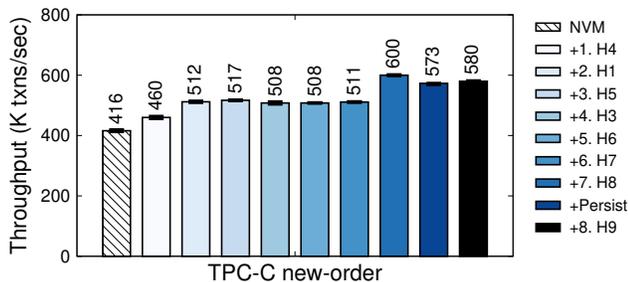


Figure 15. The contribution of optimizations to the throughput of DrTM+H for TPC-C/no. Note that the error bars are small.

database while other machines execute transactions. We use two representative OLTP benchmarks to evaluate its performance:

TPC-C/no [46] simulates the workload of an ordering system that contains complex read/write workloads. We configure the NVM server to store ten warehouses and other machines to execute the *new-order* (no) transaction, the dominant transaction of TPC-C [46].

SmallBank [45] models a simple banking application where each transaction issues one or two read/write requests. We store 20,000,000 bank accounts at the NVM server and run the standard-mix transactions at clients.

Comparing targets. In Figure 14, **DRAM** is the vanilla DrTM+H running on DRAM. **+NVM** runs DrTM+H on Optane PM, and **+OPT(H1-H8)** further applies optimizations (§6.1.1). **+Persist** adopts an existing approach [20] to support durability atop of **+OPT(H1-H8)**. Finally, **+OPT(H9)** optimizes **+Persist** with **H9**.

Performance without Persistence. Figure 14 presents the throughput-latency results of both workloads. We plot the graph by increasing the number of clients until the throughput is saturated. **+OPT(H1-H8)** improves the DrTM+H’s performance under TPC-C/no and SmallBank by 1.45X and 2.20X, respectively.

To analyze the contributions of each optimization, Figure 15 and Figure 16 further present factor analyses of evaluation results on TPC-C/no and SmallBank, respectively. First, we can see that several hints are beneficial to both workloads. For example, **H8** (use atomic operations less on NVM) speedups TPC-C/no and SmallBank by 1.17X and 1.19X, respectively. On the other hand, some hints have negative effects for certain workloads: SmallBank drops 15% throughput when adding **H3**, this is because **H3** is only beneficial when the application is bottlenecked by NVM’s bandwidth. Finally, some hints have more contributions to SmallBank than TPC-C/no. For example, **H7** has a 1.4X speedup on SmallBank but does not affect TPC-C/no. **H7** only improves transaction utilizations of NVM write throughput, while

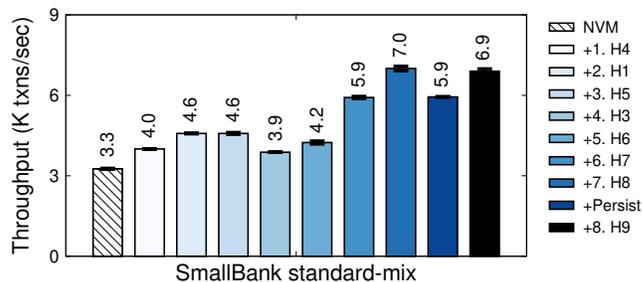


Figure 16. The contribution of optimizations to the throughput of DrTM+H for SmallBank. Note that the error bars are small.

SmallBank is more sensitive to the NVM write throughput utilization due to its simpler workloads.

Performance with persistence. As shown in Figure 14, supporting persistent transaction (**+Persist**) adds 5% and 15% performance overhead to **+OPT(H1-H8)** on TPC-C/no and SmallBank, respectively. The overhead is from the additional one-sided RDMA READ at the logging phase. Hence, reducing this network roundtrip with **H9** improves TPC-C/no and SmallBank’s performance by 1.01X and 1.17X, respectively.

6.2 Distributed file system

Octopus [32] is a distributed file system designed for RDMA and NVM. Due to space limitations, we only give a brief overview of it and our applied optimizations.

Overview. Octopus uses a distributed NVM pool to store the file system metadata and its file data blocks. It achieves high throughput and bandwidth for reading/write file data through *Client-Active Data I/O*: the client directly read/write a file’s data block using one-sided RDMA READ/WRITE. Besides *Client-Active Data I/O*, Octopus also leverages RDMA-enabled distributed transactions to update the filesystem metadata. Similar to DrTM+H (§6.1), its transactional protocol uses one-sided RDMA ATOMICs to coordinate conflicting metadata operations.

Optimizations. We focus on improving Octopus’s *Client-Active Data I/O* because the distributed transaction is not supported in its current public available codebase⁹. Nevertheless, we believe our findings can also improve distributed transaction performance in Octopus, e.g., using **H8** to improve its lock performance.

6.2.1 Evaluation

We use the same Data I/O benchmark in the Octopus paper [32] for the evaluation. In this benchmark, each client writes a fixed payload to a random location in a randomly chosen file. The client first uses two-sided RDMA to query

⁹<https://github.com/thustorage/octopus>

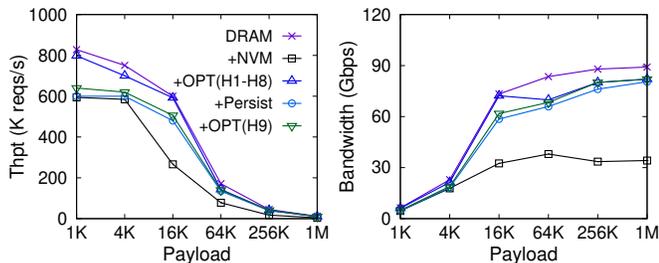


Figure 17. Data I/O (a) throughput and (b) bandwidth of Octopus (Multiple Clients).

the file metadata (e.g., data block addresses). Then, it writes the data payload with one-sided RDMA WRITE.

Comparing targets. In Figure 17, **DRAM** is the vanilla Octopus using DRAM to emulate the NVM pool. **+NVM** uses Optane PM as NVM pool, and **+OPT(H1-H8)** applies our optimizations to **+NVM**. **+Persist** further adopts an existing approach [20] to support synchronous durable file write atop pf **+OPT(H1-H8)**. Finally, **+OPT(H9)** leverages **H9** to reduce network roundtrips for persistence of **+Persist**.

Performance. As shown in Figure 17, **+OPT(H1-H8)** improve **+NVM** by up to 2.4X (from 1.2X), mainly due to applying **H3**. Without **H3**, Octopus’s client-active I/O cannot fully saturate NVM’s write bandwidth because it uses one-sided RDMA WRITE with DDIO enabled to write to the NVM. Further, **+OPT(H9)** outperforms **+Persist** by 1.06X (from 1.02X), thanks to the reduced RDMA roundtrips for persistent write.

7 Related Work

RDMA-NVM systems. We continue the line of research of using RDMA and NVM to improve the performance and reliability of distributed systems [8, 11, 32, 33, 38, 39, 43, 58, 61]. Kashyap *et al.* [27] explores the trade-offs of using different methods to ensure NVM write persistence with RDMA. They conduct their experiments on emulated NVM. Our study instead focuses on Optane PM. Orion [58] is a distributed file system designed for RDMA and NVM. It does not consider RDMA-aware optimizations (i.e., **H9**) and chooses two-sided RDMA for the persistent write. Our study provides another design decision for them. Hotpot [38] uses RDMA and NVM to build a distributed persistent shared memory. AsymNVM [33] proposes an asymmetric architecture to use NVM with RDMA. Though AsymNVM is evaluated with Optane PM, it does not consider the performance features of Optane PM. Hence, we believe our study can further improve its performance on Optane PM.

RDMA-aware optimizations. Our work is built upon broadly explored RDMA-aware optimizations [7, 10, 24, 25, 47, 52, 53]. The evaluating execution framework [53] has integrated most of these optimizations. FaRM [10] proposes

various techniques to utilize RNIC’s cache better, e.g., using huge pages. Kalia *et al.* [24] present the importance of understanding the low-level factors of how RNIC works. Based on this, they propose various RDMA-aware optimizations such as doorbell batching. FaSST [25] presents an efficient and scalable RPC framework atop two-sided RDMA datagram primitive. LITE [47] uses kernel indirection to improve the scalability of one-sided RDMA. Wukong [40, 56, 60] leverages RDMA to improve the performance of distributed graph store and further considers the interaction between RDMA and GPU [51].

NVM-aware systems. Except for RDMA-NVM systems, researchers are building NVM-aware systems for decades, including but not limited to file systems [9, 12, 57], NVM-aware data structures [48, 63], key-value stores [26], NVM-aware JVM [41, 55], and transactions on NVM [14, 15, 30, 34, 50]. Like RDMA-NVM systems, most of them use emulated NVM since there is no commercially available NVM at that time. We hope our study can further inspire future research on revisiting previous NVM-aware systems on Optane PM.

8 Conclusion

Designing high-performance RDMA-NVM systems requires a clear understanding of the interaction between RDMA and NVM. This paper provides a systematic study on how to best utilize NVM with RDMA, which summarizes nine optimization hints. By examining existing RDMA-NVM systems with these hints, we found room for improvements, especially for those not targeting production NVM: our optimized DrTM+H is up to 2.2X faster on Optane PM, while our optimized Octopus file system is up to 2.4X faster. We believe our summarized hints as well as experiences in applying them to existing systems can benefit future systems developers when designing systems with RDMA and NVM.

9 Acknowledgment

We sincerely thank our shepherd Aleksandar Dragojevic and the anonymous reviewers for their insightful comments and feedback. This work was supported in part by the National Key Research & Development Program of China (No. 2020YFB2104100), the National Natural Science Foundation of China (No. 61732010, 61925206), and a grant from Huawei Technologies. Corresponding author: Rong Chen (rongchen@sjtu.edu.cn).

References

- [1] NetCAT. <https://www.vusec.net/projects/netcat/>.
- [2] The Volatile Benefit of Persistent Memory. <https://memcached.org/blog/persistent-memory/>, 2020.

- [3] 200Gb/s ConnectX-6 Ethernet Single/Dual-Port Adapter IC. <https://www.mellanox.com/products/ethernet-adapter-ic/connectx-6-en-ic>, 2021.
- [4] Build Persistent Memory Applications with Reliability Availability and Serviceability. <https://software.intel.com/content/www/us/en/develop/articles/build-pmem-apps-with-ras.html>, 2021.
- [5] ipmctl. <https://github.com/intel/ipmctl>, 2021.
- [6] ASSOCIATION., I. T. Infiniband architecture specification. <https://cw.infinibandta.org/document/dl/7859>, 2015.
- [7] CHEN, H., CHEN, R., WEI, X., SHI, J., CHEN, Y., WANG, Z., ZANG, B., AND GUAN, H. Fast in-memory transaction processing using rdma and htm. *ACM Trans. Comput. Syst.* 35, 1 (July 2017).
- [8] 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* (New York, NY, USA, 2016), EuroSys '16, Association for Computing Machinery.
- [9] DONG, M., AND CHEN, H. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, July 2017), USENIX Association, pp. 719–731.
- [10] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 401–414.
- [11] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, Association for Computing Machinery, p. 54–70.
- [12] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, Association for Computing Machinery.
- [13] FARSHIN, A., ROOZBEH, A., JR., G. Q. M., AND KOSTIĆ, D. Reexamining direct cache access to optimize i/o intensive applications for multi-hundred-gigabit networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 673–689.
- [14] HSU, T. C.-H., BRÜGNER, H., ROY, I., KEETON, K., AND EUGSTER, P. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, Association for Computing Machinery, p. 468–482.
- [15] HU, Q., REN, J., BADAM, A., SHU, J., AND MOSCIBRODA, T. Log-structured non-volatile main memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, July 2017), USENIX Association, pp. 703–717.
- [16] INTEL. Intel® data direct i/o technology. <https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html>, 2019.
- [17] INTEL. Intel® memory latency checker v3.7. <https://software.intel.com/en-us/articles/intelr-memory-latency-checker>, 2019.
- [18] INTEL. Intel® xeon® processor e5 v4 product family. <https://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-v4-datasheet-vol-2.html>, 2019.
- [19] INTEL. Persistent Memory Replication Over Traditional RDMA. <https://software.intel.com/en-us/articles/persistent-memory-replication-over-traditional-rdma-part-1-understanding-remote-persistent>, 2019.
- [20] INTEL. The librpmem library. <https://pmem.io/pmdk/librpmem/>, 2019.
- [21] INTEL. Intel® Optane persistent memory 200 series. <https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html>, 2020.
- [22] KALIA, A., ANDERSEN, D., AND KAMINSKY, M. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (New York, NY, USA, 2020), SoCC '20, Association for Computing Machinery, p. 105–119.
- [23] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, Association for Computing Machinery, p. 295–306.
- [24] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, June 2016), USENIX Association, pp. 437–450.
- [25] 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 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 185–201.
- [26] KANNAN, S., BHAT, N., GAVRILOVSKA, A., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. Redesigning LSMs for nonvolatile memory with NovelSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 993–1005.
- [27] KASHYAP, S., QIN, D., BYAN, S., MARATHE, V. J., AND NALLI, S. Correct, fast remote persistence. *arXiv preprint arXiv:1909.02092* (2019).

- [28] KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213–226.
- [29] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 429–444.
- [30] LIU, M., ZHANG, M., CHEN, K., QIAN, X., WU, Y., ZHENG, W., AND REN, J. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2017), ASPLOS '17, Association for Computing Machinery, p. 329–343.
- [31] LIU, X., HUA, Y., LI, X., AND LIU, Q. Write-optimized and consistent RDMA-based NVM systems. *arXiv preprint arXiv:1906.08173* (2019).
- [32] LU, Y., SHU, J., CHEN, Y., AND LI, T. Octopus: an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, July 2017), USENIX Association, pp. 773–785.
- [33] MA, T., ZHANG, M., CHEN, K., SONG, Z., WU, Y., AND QIAN, X. Asymnvm: An efficient framework for implementing persistent data structures on asymmetric NVM architecture. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2020), ASPLOS '20, Association for Computing Machinery, p. 757–773.
- [34] MEMARIPOUR, A., BADAM, A., PHANISHAYEE, A., ZHOU, Y., ALAGAPPAN, R., STRAUSS, K., AND SWANSON, S. Atomic in-place updates for non-volatile main memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, Association for Computing Machinery, p. 499–512.
- [35] NEUGEBAUER, R., ANTICHI, G., ZAZO, J. F., AUDZEVICH, Y., LÓPEZ-BUEDO, S., AND MOORE, A. W. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, Association for Computing Machinery, p. 327–341.
- [36] RYAN SMITH. 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.
- [37] SHAMIS, A., RENZELMANN, M., NOVAKOVIC, S., CHATZOPOULOS, G., DRAGOJEVIĆ, A., NARAYANAN, D., AND CASTRO, M. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 433–448.
- [38] SHAN, Y., TSAI, S.-Y., AND ZHANG, Y. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY, USA, 2017), SoCC '17, Association for Computing Machinery, p. 323–337.
- [39] SHEN, S., CHEN, R., CHEN, H., AND ZANG, B. Retrofitting High Availability Mechanism to Tame Hybrid Transaction-/Analytical Processing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)* (July 2021), USENIX Association.
- [40] SHI, J., YAO, Y., CHEN, R., CHEN, H., AND LI, F. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 317–332.
- [41] SHULL, T., HUANG, J., AND TORRELLAS, J. Autopersist: An easy-to-use Java NVM framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2019), PLDI 2019, Association for Computing Machinery, p. 316–332.
- [42] SMOLYAR, I., MARKUZE, A., PISMENNY, B., ERAN, H., ZELLWEGER, G., BOLEN, A., LISS, L., MORRISON, A., AND TSAFRIR, D. Ioctopus: Outsmarting nonuniform dma. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2020), ASPLOS '20, Association for Computing Machinery, p. 101–115.
- [43] TALEB, Y., STUTSMAN, R., ANTONIU, G., AND CORTES, T. Tailwind: Fast and atomic rdma-based replication. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 851–863.
- [44] TALPEY, T., AND KAMER, G. High performance file serving with SMB3 and RDMA via SMB direct. In *Storage Developers Conference* (2012).
- [45] THE H-STORE TEAM. SmallBank Benchmark. <http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/>.
- [46] THE TRANSACTION PROCESSING COUNCIL. TPC-C Benchmark V5.11. <http://www.tpc.org/tpcc/>.
- [47] TSAI, S.-Y., AND ZHANG, Y. LITE kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 306–324.
- [48] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies* (USA, 2011), FAST'11, USENIX Association, p. 5.
- [49] VOLOS, H., MAGALHAES, G., CHERKASOVA, L., AND LI, J. Quartz: A lightweight performance emulator for persistent memory software. In *Proceedings of the 16th Annual Middleware Conference* (New York, NY, USA, 2015), Middleware '15, Association for Computing Machinery, p. 37–49.

- [50] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, Association for Computing Machinery, p. 91–104.
- [51] WANG, S., LOU, C., CHEN, R., AND CHEN, H. Fast and concurrent RDF queries using rdma-assisted GPU graph exploration. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 651–664.
- [52] WEI, X., CHEN, R., AND CHEN, H. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 117–135.
- [53] 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)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 233–251.
- [54] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, Association for Computing Machinery, p. 87–104.
- [55] WU, M., ZHAO, Z., LI, H., LI, H., CHEN, H., ZANG, B., AND GUAN, H. Espresso: Brewing java for more non-volatility with non-volatile memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2018), ASPLOS '18, Association for Computing Machinery, p. 70–83.
- [56] XIE, X., WEI, X., CHEN, R., AND CHEN, H. Pragh: Locality-preserving graph traversal with split live migration. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 723–738.
- [57] 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)* (Santa Clara, CA, Feb. 2016), USENIX Association, pp. 323–338.
- [58] YANG, J., IZRAELEVITZ, J., AND SWANSON, S. Orion: A distributed file system for non-volatile main memory and RDMA-Capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 221–234.
- [59] YANG, J., KIM, J., HOSEINZADEH, M., IZRAELEVITZ, J., AND SWANSON, S. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 169–182.
- [60] ZHANG, Y., CHEN, R., AND CHEN, H. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17, p. 614–630.
- [61] ZHANG, Y., YANG, J., MEMARIPOUR, A., AND SWANSON, S. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, Association for Computing Machinery, p. 3–18.
- [62] ZIEGLER, T., TUMKUR VANI, S., BINNIG, C., FONSECA, R., AND KRASKA, T. Designing distributed tree-based index structures for fast rdma-capable networks. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 741–758.
- [63] ZUO, P., HUA, Y., AND WU, J. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 461–476.

MigrOS: Transparent Live-Migration Support for Containerised RDMA Applications

Maksym Planeta
TU Dresden

Jan Bierbaum
TU Dresden

Leo Sahaya Daphne Antony
AMOLF

Torsten Hoefler
ETH Zürich

Hermann Härtig
TU Dresden

Abstract

RDMA networks offload packet processing onto specialised circuitry of the network interface controllers (NICs) and bypass the OS to improve network latency and bandwidth. As a consequence, the OS forfeits control over active RDMA connections and loses the possibility to migrate RDMA applications transparently. This paper presents MigrOS, an OS-level architecture for transparent live migration of containerised RDMA applications. MigrOS shows that a set of minimal changes to the RDMA communication protocol reenables live migration without interposing the critical path operations. Our approach requires no changes to the user applications and maintains backwards compatibility at all levels of the network stack. Overall, MigrOS can achieve up to 33% lower network latency in comparison to software-only techniques.

1 Introduction

Major cloud providers increasingly offer RDMA network connectivity [1, 55] and high-performance network stacks [8, 18, 37, 53, 69, 85, 89] to the end-users. RDMA networks lower communication latency and increase network bandwidth by offloading packet processing to the RDMA NICs and by removing the OS from the communication critical path. To remove the OS, user applications employ specialised RDMA APIs, which access RDMA NICs directly, when sending and receiving messages [54]. These performance benefits made RDMA networks ubiquitous both in the HPC [15, 30, 43, 50, 77] and in the cloud settings [22, 59, 72].

At the same time, containers have become a popular tool for lightweight virtualisation. Containerised applications, being independent of the host’s user space (libraries, applications, configuration files), greatly simplify distributed application deployment and administration. However, RDMA networks and containers follow contradicting architectural principles: Containerisation enforces stricter isolation between applications and the host, whereas RDMA networks try to bring applications and underlying hardware “closer” to each other.

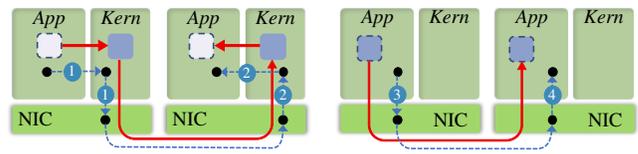


Figure 1: Traditional (left) and RDMA (right) network stacks. In traditional networks, the user application triggers the NIC via kernel (1). After receiving a packet, the NIC notifies the application back through the kernel (2). In RDMA networks the application communicates directly to the NICs (3) and vice-versa (4) without kernel intervention. Traditional networks require copy between application buffers (□) and NIC accessible kernel buffers (■). RDMA NICs (right) access message buffers in the application memory directly (■).

Container orchestration systems, like Kubernetes, stop containers and restart them on other hosts for the purpose of load balancing, resiliency, and administration. To maintain efficiency, containerised applications are expected to restart quickly [9]. Unfortunately, fast application restart requires in-application support and can be very costly, as RDMA applications tend to have large state. In contrast, *live migration* moves application state transparently and imposes no additional requirements for the application.

In the context of live migration, much of the container state resides in the host OS kernel. It can be extracted and recovered elsewhere later on. This recoverable state includes open TCP connections, shell sessions, file locks [13, 40]. However, as we elaborate in Section 3, *the state of RDMA communication channels is not recoverable by existing systems, and hence RDMA applications cannot be checkpointed or migrated.*

To outline the conceptual difficulties involved in saving the state of RDMA communication channels, we compare a traditional TCP/IP-based network stack and the IB verbs API¹ (see Figure 1). First, with a traditional network stack, the kernel fully controls when the communication happens:

¹ IB verbs is the most common low-level API for RDMA networks.

applications need to perform system calls to send or receive messages. In IB verbs, because of the direct communication between the NIC and the application, the OS cannot alter the connection state, except for tearing the connection down. Although the OS can stop a process from sending further messages, the NIC may still silently change the application state. Second, part of the connection state resides at the NIC and is inaccessible for the OS. Creating a consistent checkpoint transparently is impossible in this situation.

The contribution of our paper is a way to overcome the described limitations of current RDMA networks and the supporting operating systems with a novel architecture. The root of the problem to be solved is the impossibility for the OS to intercept packets to manipulate and repair disrupted connections, as it happens in transparent live application migration. The core point of the architecture is a small change in the RDMA protocol, that is usually implemented in RDMA NICs: we propose to add two new states and two new message types to RoCEv2, a popular RDMA communication protocol, while paying careful attention to backwards compatibility.

To make the proposed changes credible, we show that they are 1. *sufficient* for migrations, as an example for disrupted connections, and 2. indeed *small*. To prove the first point, we design and implement a transparent end-to-end live migration architecture for containerised RDMA applications. To prove the second point, and to enable the software architecture, we implement the new protocol by modifying SoftRoCE [48], a software implementation of the RoCEv2 protocol.

Providing a credible evaluation is hard for us, since we cannot modify the state machine of an RDMA NIC. Instead, we substantiate our claims by comparing the latency and bandwidth of the original and our modified protocol using SoftRoCE implementations and show that outside of the migration phase network performance is not affected. To justify the proposed protocol changes and the container-based software architecture, we evaluate software-level support for transparent live migration [2, 44]. We show that these approaches add significant overhead to the *normal* operation.

2 Background

This section gives a short introduction to containerisation and RDMA networking. We further outline live migration and how RDMA networking obstructs this process.

2.1 Containers

In Linux, processes can be logically separated from the rest of the system using *namespaces*. This way processes can have an isolated view on the file system, network devices, users, etc. Container runtimes leverage namespaces and other low-level kernel mechanisms [17, 47] to create a complete system view inside a *container* with one or multiple processes. Migration of a container moves all processes running inside

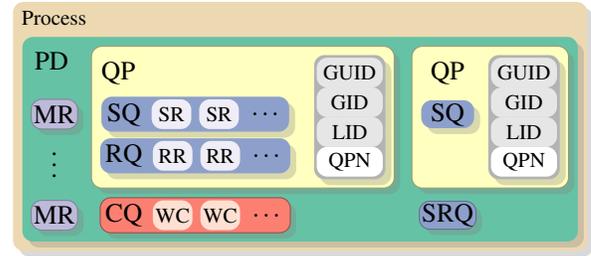


Figure 2: Primitives of the IB verbs library. Each QP comprises a send and a receive queue and has multiple IDs; node-global IDs (grey) are shared by all QPs on the same node.

a given container. A distributed application may comprise multiple containers across a network: a Spark application, for example, can isolate each *worker* in a container and an MPI [21] application can containerise each *rank*.

2.2 Infiniband verbs

The IB verbs API is a de-facto standard for high-performance RDMA communication today. RDMA applications achieve high throughput and low latency by accessing the NIC directly (*OS-bypass*), reducing memory movement (*zero-copy*), and delegating packet processing to the NIC (*offloading*).

Figure 2 shows the following IB verbs objects involved in communication. *Memory regions* (MRs) represent pinned memory shared between the application and the NIC. *Queue pairs* (QPs), comprising a send queue (SQ) and a receive queue (RQ), represent connections. To reduce the memory footprint, the individual RQs of multiple QPs can be replaced with a single *shared receive queue* (SRQ). *Completion queues* (CQs) inform the application about completed communication requests. A *protection domain* (PD) groups all these objects together and represents the process address space to the NIC.

To establish a connection, the IB verbs specification [54] requires the applications to exchange the following addressing information: *Memory protection keys* to enable access to remote MRs, the global vendor-assigned address (*GUID*), the routable address (*GID*), the non-routable address (*LID*), and the node-specific QP number (*QPN*). One way to perform this exchange is over an out-of-band network, e.g. Ethernet. During the connection setup, each QP is configured for a specific *type of service*. MigrOS supports only Reliable Connections (RC), which provide reliable in-order message delivery between two communication partners. Another popular type of service is Unreliable Datagram (UD), which does not provide these guarantees.

The application sends or receives messages by posting *send requests* (SR) or *receive requests* (RR) to a QP as *work queue entries* (WQE). These requests describe the message and refer to the memory buffers within previously created MRs. The application checks for the completion of outstanding work

requests by polling the CQ for *work completions* (WC).

There are various implementations of the IB verbs API for different hardware, including Infiniband [37], iWarp [23], and RoCE [35, 36]. InfiniBand is generally the fastest among these networks but requires specialised NICs and switches. RoCE and iWarp are easier to deploy, because they provide RDMA capabilities in Ethernet networks. They still require hardware support in the NIC, but do not depend on specialised switches. This work focuses on RoCEv2, an increasingly more popular version of the RoCE protocol [15, 60, 88]. At the same time, we change only parts of RoCEv2 that are defined in the same way for Infiniband [37] and RoCEv1 [35], hence MigrOS is also compatible with other RDMA protocols.

To enable RDMA-application migration, it is important to consider the following challenges:

1. User applications have to use physical network addresses (QPN, LID, GID, GUID) but the IB verbs API does not specify a way for virtualising these.
2. The NIC can write to any memory it shares with the application without the OS noticing.
3. The OS cannot instruct the NIC to pause communication, except for abruptly terminating it.
4. User applications do not handle changes in a connection destination address and will go into an erroneous state. As a result, the application will terminate abruptly.
5. Although the OS resides on the control path and is therefore aware of all IB verbs objects created by the application, the OS does not control the whole state of these objects, as the state partially resides on the NIC.

We address all of these challenges in [Section 3](#).

2.3 CRIU

CRIU is a software framework for transparent checkpointing and restoring of Linux processes [13]. It enables live migration, snapshots, or accelerated start-up of processes and containers. To extract the user-space application state, CRIU uses conventional debugging mechanisms, like ptrace [73, 74]. However, to extract the state of process-specific kernel objects, CRIU depends on special Linux kernel interfaces.

During recovery, CRIU runs inside the target process and recreates all OS objects on behalf of the target. This way, CRIU utilises the available OS mechanisms to run most of the recovery without the need for significant kernel modifications. Finally, CRIU removes any traces of itself from the process.

CRIU can also restore the state of TCP connections, a crucial feature for live migration of distributed applications [40]. The Linux kernel introduced a new TCP connection state, `TCP_REPAIR`, for that purpose. In this state, a user-level process can modify the send and receive message queues, get and set the message sequence numbers and timestamps, or open and close a connection without notifying the other side.

As of now, CRIU does not support saving and restoring IB verbs objects. Discarding IB verbs objects during migra-

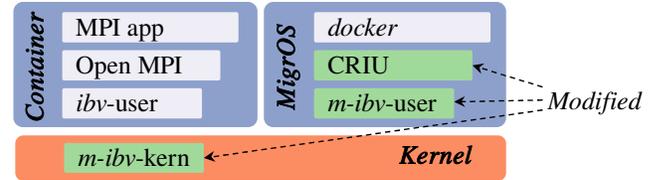


Figure 3: MigrOS architecture. Software inside the container, including the user-level driver (*ibv-user*, grey), is unmodified. The host runs CRIU, kernel- (*m-ibv-kern*) and user-level (*m-ibv-user*, green) drivers modified for migratability.

tion in the naive hope that the application will be able to recover is failure-prone: once an application runs into an erroneous IB verbs object, in most cases, the application will hang or crash. Thus, MigrOS provides explicit support for IB verbs objects in CRIU (see [Section 3](#)).

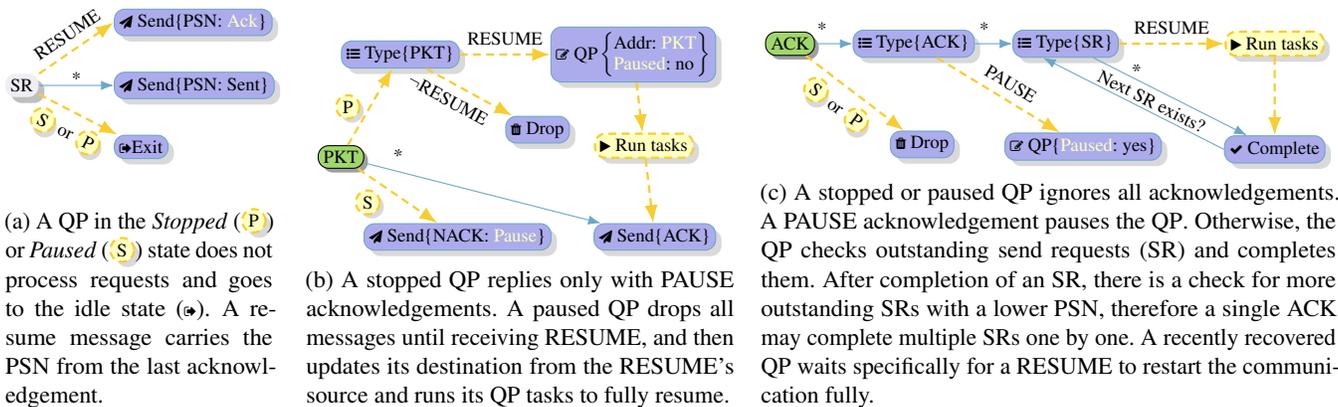
3 Design

MigrOS is based on modern container runtimes and reuses much of the existing infrastructure with minimal changes (see [Section 3.1](#)). Most importantly, we require no modification of the software running inside the container. Instead, MigrOS includes the following changes concerning RoCEv2: two new QP states to enable the creation of consistent checkpoints (see [Section 3.2](#)), a connection migration protocol ([Section 3.3](#)), and modifications to the packet-level RoCEv2 protocol ([Section 3.4](#)). Finally, [Section 3.5](#) describes our modifications to the IB verbs API and how CRIU uses them. It is sufficient to only extend CRIU with IB verbs support, existing container runtimes use CRIU for their checkpoint/restore functionality [17, 47, 71, 76].

3.1 Software Stack

Typically, access to the RDMA network is hidden deep inside the software stack. [Figure 3](#) gives an example of a containerised RDMA application. The container image comes with all library dependencies, like the `libc`, but not the kernel-level drivers. In this example, the application uses a stack of communication libraries, comprising Open MPI [21], Open UCX [77] (not shown), and IB verbs. Normally, to migrate, a container runtime would require the application inside the container to terminate and later recover all IB verbs objects. This removes transparency from live migration.

MigrOS runs alongside the container comprising a container runtime (e.g., Docker [17]), CRIU, and the IB verbs library. We modified CRIU to make it aware of IB verbs, so it can save IB verbs objects when traversing the kernel objects that belong to the container. We extend the IB verbs library (*m-ibv-user* and *m-ibv-kern*) to enable serialisation and deserialisation of the IB verbs objects. Importantly, the



(a) A QP in the *Stopped* (P) or *Paused* (S) state does not process requests and goes to the idle state (⊙). A resume message carries the PSN from the last acknowledgement.

(b) A stopped QP replies only with PAUSE acknowledgements. A paused QP drops all messages until receiving RESUME, and then updates its destination from the RESUME's source and runs its QP tasks to fully resume.

(c) A stopped or paused QP ignores all acknowledgements. A PAUSE acknowledgement pauses the QP. Otherwise, the QP checks outstanding send requests (SR) and completes them. After completion of an SR, there is a check for more outstanding SRs with a lower PSN, therefore a single ACK may complete multiple SRs one by one. A recently recovered QP waits specifically for a RESUME to restart the communication fully.

Figure 6: MigrOS changes processing of incoming packets (PKT and ACK) and outgoing request (SR) by introducing new QP states (P and S), new messages (PAUSE and RESUME), new operations (⊙ and ▶), and new transitions (-▶). Solid arrows (→) represent previously existing transitions, “*” represents the “else” branch, ≡ Type checks the SR or packet type.

run in hardware. We proceed from the fact that a typical RDMA NIC does all packet-level work, including transmitting packets and acknowledgements by itself. As a consequence, the OS cannot itself compose and process arbitrary packets, including the packets that MigrOS introduces.

Therefore, for migration to work, we need to modify RDMA NICs. The NIC adds different packet headers, depending on the service (RC, UD, etc.) and message (read, send, etc.) type. Our changes touch only two headers: Base Transport Header (BTH) and ACK Extended Transport Header (AETH). BTH is the first RDMA-specific header, immediately following IP and UDP headers. Additionally, the NIC needs to maintain two new flags per QP for pause and resume states. The NIC must be able to transfer a QP into the *Stopped* or *Paused* state, process a send request with a resume message issued by the OS, and handle pause/resume packets.

Our changes (see Figure 6) cover three existing workflows in the underlying RoCEv2 protocol: 1. Processing a send request WQE (Figure 6a), 2. receiving a packet (Figure 6b), and 3. receiving an acknowledgement (Figure 6c). The NIC must consider the two new states of the QP in all these workflows. This change of logic is small and does not change the packet layout, as it is only part of the internal state of the QP.

The NIC must compose the new resume message or let the OS do so. In contrast to a normal message, resume takes the packet sequence number (PSN) from the last acknowledged message instead of the last sent message. A receiver recognises the resume message by a new *opcode* in the BTH header. Creating a new message type does not require changes in the message layout due to an abundance of unused opcodes.

Similarly, pause, sent as a new negative acknowledgement type, employs an unused value of the *syndrome* field in the AETH header. Therefore, the new pause NACK also requires no change to the existing packet layout.

The workflows in the Figure 6 run when the user triggers

the NIC through a doorbell register, when a message arrives, or by a timeout. Unless a QP is paused or stopped, the NIC will try to send or complete multiple messages at once (Figure 6a and Figure 6c). As part of resume (▶), the NIC also triggers these workflows.

Figure 6 does not include, for example, additional timeout reaction logic for the sake of brevity. Overall, the changes in logic are simple and mostly reuse existing functionality. Because we change only the AETH and BTH headers, our changes are equally applicable to other RDMA protocols (e.g. Infiniband or RoCEv1) that use these headers in the same way. We believe neither new logic nor new states incur prohibitive design or implementation cost.

A real RDMA NIC would need hardware support for the new QP states and follow the corresponding protocol. Full migration support would also require the NIC to extract the state of Infiniband objects and handle the new message types. We believe all of this can be implemented in the NIC's firmware². Section 4 presents a proof-of-concept software implementation of the proposed changes.

3.5 Checkpoint/Restore API

To enable checkpoint/restore for processes and containers, we extend the IB verbs API with two new calls (see Listing 1): `ibv_dump_context` and `ibv_restore_object`. CRIU relies on the normal IB verbs API supplemented by the two new calls to save and restore the IB verbs state of applications.

The `ibv_dump_context` call returns a dump of all IB verbs objects within a specific IB verbs *context*, an object representing the connection between a process and an RDMA NIC. The creation of a dump runs almost entirely inside the kernel for two reasons: First, some links between the objects are

²The hardware/firmware-boundary will differ for FPGA and ASIC.

```

int ibv_dump_context(struct ibv_context *ctx,
                    int *count, void *dump, size_t length);
int ibv_restore_object(struct ibv_context *ctx,
                      void **object, int object_type, int cmd,
                      void *args, size_t length);

```

Listing 1: Checkpoint/Restore extension for the IB verbs API. `ibv_dump_context` creates an image of the IB verbs context `ctx` with `count` objects and stores it in the caller-provided memory region `dump` of size `length`. `ibv_restore_object` executes the restore command `cmd` for an individual object (QP, CQ, etc.) of type `object_type`. The call expects a list of arguments specific to the object type and recovery command. `args` is an opaque pointer to the argument buffer of size `length`. A pointer to the restored object is returned via `object`.

only visible at the kernel level. Second, to get a consistent checkpoint it is crucial to ensure the dump is atomic.

Although the existing IB verbs API allows to create new objects, it is not expressive enough for *restoring* them. For example, when restoring a completion queue (CQ), the current API does not allow to specify the address of the shared memory region for this queue, instead this address is assigned by the kernel. It is also not possible to recreate a queue pair (QP) directly in its original state, like Ready-to-Send (RTS). Instead, the QP has to traverse all intermediate states to reach the desired state.

We introduce the fine-grained `ibv_restore_object` call to restore IB verbs objects one by one, for situations when the existing API is not sufficient. During recovery, CRIU reads the object dump and applies a specific recovery procedure for each object type. For example, to recover a QP, CRIU calls `ibv_restore_object` with the command `CREATE` and transitions the QP through the `Init`, `RTR`, and `RTS` states using `ibv_modify_qp`. The memory regions or QP buffers are recovered using the standard file and memory operations. Finally, when a QP reaches the `RTS` state (representing an active connection), the new host executes the `REFILL` command using the `ibv_restore_object` call. This command restores the driver-specific internal QP state and sends a *resume* message to the partner QP.

4 Implementation

To provide transparent live migration, MigrOS makes changes to CRIU, the IB verbs library, the RDMA-device driver (SoftRoCE), and the packet-level RoCEv2-protocol. To migrate an application, the container runtime invokes CRIU which checkpoints the target container. CRIU stops active RDMA connections and saves the state of IB verbs objects (see [Section 4.1](#)). SoftRoCE then pauses communication using our extensions to the packet-level protocol. After transferring the checkpoint to the destination node, the container runtime at that node invokes CRIU to recover the IB verbs objects and

restores the application. SoftRoCE then resumes all paused communication to complete the migration process.

SoftRoCE is a Linux kernel-level software implementation (not an emulation [48]) of the RoCEv2 protocol [36]. RoCEv2 runs RDMA communication by tunnelling Infiniband packets through a well-known UDP port. In contrast to other RDMA-device drivers, SoftRoCE allows the OS to inspect, modify, and control the state of IB verbs objects completely.

As a performance-critical component of RDMA communication, RoCEv2 usually runs inside the hardware and firmware of a NIC. We focus on minimising these protocol changes. The key part of MigrOS is the addition of connection migration capabilities to the existing RoCEv2 protocol (see [Section 4.2](#)).

4.1 State Extraction and Recovery

State extraction begins when CRIU discovers that its target process has opened an IB verbs device. We modified CRIU to use the API presented in [Section 3.5](#) to extract the state of all available IB verbs objects. CRIU stores this state together with other process data in an image. Later, CRIU recovers the image on another node using the new API.

When CRIU recovers MRs and QPs of the migrated application, the recovered objects must maintain their original unique identifiers. These identifiers are system-global and assigned by the NIC (in our case the SoftRoCE driver) in a sequential manner. We augmented the SoftRoCE driver to expose the IDs of the last assigned MR and QP to MigrOS in userspace. These IDs are *memory region number* (MRN) and *queue pair number* (QPN) correspondingly. Before recreating an MR or QP, CRIU configures the last ID appropriately. If no other MR or QP occupies this ID, the newly created object will maintain its original ID. This approach is analogous to the way CRIU maintains the process ID of a restored process using the `ns_last_pid` mechanism of Linux, which exposes the last process ID assigned by the kernel.

It is possible for some other process to occupy an MRN or QPN, which CRIU wants to restore. Two processes cannot use the same MRN or QPN on the same node, resulting in a conflict. In the current scheme, we avoid these conflicts by partitioning QP and MR addresses globally among all nodes in the system before application startup. CRIU faces the very same problem with process ID collisions. This problem has only been solved with the introduction of process ID namespaces. To remedy the collision problem for IB verbs objects, a similar namespace-based mechanism, together with a virtual RDMA network [29], would be required. We leave this issue for future work.

Additionally, recovered MRs have to maintain their original memory protection keys. The protection keys are pseudo-random numbers [75] provided by the NIC and are used by a remote communication partner when sending a packet. An RDMA operation succeeds only if the provided key matches

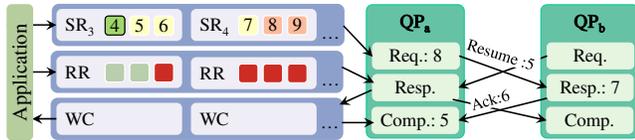


Figure 7: Resuming a connection in SoftRoCE. The figure depicts a snapshot of an immediate state, whereas the arrows indicate the flow of data over time. A send queue comprises multiple SRs, each expected to send multiple packets. Packets 8 and 9 (●) are to be processed by the requester. Packets 5–7 (●) are yet to be acknowledged. Packet 4 (●) is already acknowledged. A receive queue contains RRs with received (●) and not yet received (●) packets. QP_b expects the next packet to be 7. A resume packet has the PSN of the first unacknowledged packet (●). QP_b replies with an acknowledgement of the last received packet.

the expected key of a given MR. Other than that, the key's value does not carry any additional semantics. Thus, no collision problems exist for protection keys.

CRIU sets all protection keys to their original values before communication restarts by making an `ibv_restore_object` call with the `IBV_RESTORE_MR_KEYS` command.

4.2 Resuming Connections

The connection migration protocol ensures that connections are terminated gracefully and recovered to a consistent state. The implementation of this protocol is device- and driver-specific. In this work, we modify the SoftRoCE driver to make it compliant with the connection migration protocol (Section 3.3) by providing an implementation of the checkpoint/restore API (Section 3.5).

Figure 7 outlines the basic operation of the SoftRoCE driver, which creates three concurrent *tasks* for each QP: *requester*, *responder*, and *completer*. When an application posts send (SR) and receive (RR) work requests to a QP, they are processed by requester and responder correspondingly. A work request may be split into multiple packets, depending on the MTU size. When the whole work request is complete, responder or completer notify the application by posting a work completion to the completion queue.

The tasks process all requests packet by packet. Each task maintains the packet sequence number (PSN) of the next packet. A requester sends packets for processing at the responder of its partner QP. The responder replies with an acknowledgement sent to the completer. The completer generates a work completion (WC) after receiving an acknowledgement for the last packet in an SR. Similarly, the responder generates a WC after receiving all packets of an RR.

After migration, when the recovered QP_a is ready to communicate again, it sends a resume message to QP_b with the new address. Upon receiving the resume message, the re-

sponder of QP_b learns the new location of QP_a. Then, the responder replies with an acknowledgement of the last successfully received packet. If some packets were lost during the migration, the next PSN at the responder of QP_b is smaller than the next PSN at the requester of QP_a. The difference corresponds to the lost packets. Simultaneously, the requester of QP_b can already start sending messages. At this point, the connection between QP_a and QP_b is fully recovered.

The presented protocol ensures that both QPs recover the connection without losing packets irrecoverably. If packets were lost during migration, the QPs can determine which packets were lost and retransmit them, as part of the normal RoCEv2 protocol. The whole connection migration protocol runs transparently for the user applications.

5 Evaluation

We evaluate MigrOS from three main aspects. First, we analyse the implementation effort, with a specific focus on the magnitude of changes to the RoCEv2 protocol. Second, we study the overhead of adding migration capability, outside the migration phase. Third, we estimate the fine-grained cost of migration for individual IB verbs objects, as well as the full latency of migration in realistic RDMA applications.

For most experiments, we use a system with two machines: Each machine is equipped with an Intel i7-4790 CPU, 16 GiB RAM, an on-board Intel 1 Gb Ethernet adapter, a Mellanox ConnectX-3 VPI adapter, and a Mellanox Connect-IB 56 Gb adapter. The Mellanox VPI adapters are set to 40 Gb Ethernet mode. The SoftRoCE driver communicates over this adapter. The machines run Debian 11 with a custom Linux 5.7-based kernel. We refer to this setup as *local*. When comparing against DMTCP and FreeFlow, we use Ubuntu 14.04.

We conduct further measurements on a cluster comprising two-socket Intel E5-2680 v3 CPUs nodes with Connect-IB 56 Gb NICs deployed by Bull. We refer to this setup as *cluster*. Two nodes similar to those in the cluster were used in a local setup and equipped with Mellanox ConnectX-3 VPI NICs configured to 56 Gb InfiniBand mode.

5.1 Size of Changes

To quantify the changes MigrOS requires, we count the newly added or modified source lines of code (SLOC) in different components of the software stack. Out of around 4 kSLOC only around 10% apply to the kernel-level SoftRoCE driver. These changes mostly focus on saving and restoring the state of IB verbs objects. We separately counted changes to the requester, responder, and completer QP tasks responsible for the active phase of communication (see Figure 7). These tasks would be implemented in the NICs, for hardware-based RDMA implementations. Therefore, we keep the changes to QP tasks simple and small, as these changes must be reflected

Object	Features required	State (bytes)
PD	None	12
MR	Set memory keys and MRN	48
CQ	Restore ring buffer metadata	64
SRQ	Restore ring buffer metadata	68
QP	+ QP tasks state, set QPN	271
QP w/ SRQ	+ Current WQE state	823

Table 1: Additional features implemented in the kernel-level SoftRoCE driver to enable recovery of IB verbs objects. We provide the size that each object occupies in the dump.

in firmware or hardware. In our implementation, changes to QP tasks accounted only for around 6% of overall changes.

Using `gconv` [20], we verified that most of the changes to the QP tasks do not affect the critical path of communication. During the active phase of communication, 1213 lines were touched out of 4808 lines of the SoftRoCE driver identified by `gconv` as executable. Among the touched lines only 28 lines correspond to code we added for migration, with 3 lines corresponding to variable assignments and the rest being jumps related to checking for the *Paused* or *Stopped* state. The remaining code changes to the QP tasks run only during the connection migration phase.

Besides additional logic in the QP tasks, saving and restoring IB verbs objects requires the manipulation of implementation-specific attributes. Some of these attributes cannot be set through the original IB verbs API. For example, recovering an MR requires the additional ability to restore the original values of memory keys and the MRN. Some other attributes are not visible in the original IB verbs API at all. The queues (CQ, SRQ, QP) implemented in SoftRoCE require the ability to save and restore metadata of ring buffers backing up the queues. If a QP uses a shared receive queue (SRQ), the dump of the QP additionally includes the full state of the current *work queue entry* (WQE). We identified all required attributes for SoftRoCE, calculated their memory footprint (see Table 1), and implemented all features required by these attributes.

The changes to RoCEv2 implemented in SoftRoCE are small and affect the critical path of communication only marginally outside of the migration phase. We believe that for RDMA NICs the same changes to RoCEv2 will remain equally small.

5.2 Overhead of Migratability

Just adding the capability for transparent container migration already may incur overhead even when no migration occurs. For example, DMTCP (see Section 6) intercepts all IB verbs library calls and rewrites both work requests and completions before forwarding them to the NIC. Both with DMTCP and FreeFlow, this interception happens persistently, even if

Size, B	Latency, μ s			Bandwidth, Gb/s		
	Unmod.	FF	DMTCP	Unmod.	FF	DMTCP
2^0	0.8*	1.2*	1.4*	0.09	0.02	0.01
2^4	0.8*	1.2*	1.4*	1.41	0.24	0.20
2^8	1.1*	1.6*	1.8*	22.31	3.95	3.25
2^{12}	2.3	2.7	2.9	36.50	36.57	36.49
2^{16}	15.8	16.2	16.5	36.59	36.59	36.59
2^{20}	230.8	231.2	231.4	36.59	36.59	36.59

Table 2: Performance of CX3/40. Comparing execution without modifications against DMTCP and FreeFlow. The variation over 30 runs was small, except, * when $0.05 < \sigma/\mu < 0.1$.

Size, B	Latency, $\mu \pm \sigma \mu$ s		
	Plain	Migratable	DMTCP
2^0	25.3 ± 0.2	25.0 ± 0.2	26.2 ± 0.6
2^4	25.3 ± 0.3	24.9 ± 0.4	25.5 ± 0.5
2^8	26.9 ± 0.6	25.7 ± 0.7	28.0 ± 0.5
2^{12}	35.7 ± 0.4	36.8 ± 0.7	35.5 ± 0.5
2^{16}	93.2 ± 1.9	93.8 ± 1.9	94.4 ± 1.9
2^{20}	793.9 ± 12.8	802.0 ± 11.3	802.1 ± 13.5

Table 3: Communication latency with SoftRoCE. Comparison of migratable and plain (non-migratable) SoftRoCE against DMTCP using plain SoftRoCE.

the process never migrates. In contrast to this, MigrOS does not intercept communication operations on the critical path, thereby introducing no measurable overhead. This subsection explores the overhead added for normal communication operations without migrations.

DMTCP [2] and FreeFlow [44] do not offer live migration. Nevertheless, they could be extended to provide it. Therefore, we start by estimating the cost of adding migration capability at the user level. We use the latency and bandwidth benchmarks from the `perftest` benchmark suite [68]. We ran each experiment 30 times with 10000 iterations each at the *local* setup, with CX3/40 NICs.

Both frameworks do additional processing for each IB verbs work request, which results into near-constant overhead to latency (see Table 2). Each work request corresponds to a single message, not a single packet, therefore the overhead diminishes for larger message sizes. Table 2 demonstrates that bandwidth is directly affected by the increased latency and thus is lower only for small messages. We expect such bandwidth reduction to be a minor disadvantage for realistic applications, whereas a near 50% increase in latency may be critical for many latency-sensitive applications [30, 78].

It is possible, that despite our best effort to minimise the changes to the NIC, changes proposed by MigrOS introduce measurable overhead. Therefore, we need to show that Mi-

Short	Full name	Location
SR	SoftRoCE	local
CX3/40	ConnectX-3 40 Gb Ethernet	local
CX3/56	ConnectX-3 56 Gb InfiniBand	cluster
CIB	ConnectIB	local
BIB	Bull Connect-IB	cluster

Table 4: RDMA-capable NICs used for the evaluation.

grOS adds no additional cycles during message transmission. For that, we compare the performance of migratable and non-migratable versions of the SoftRoCE driver³ against DMTCP running with the non-migratable version of SoftRoCE. Running `ib_send_lat` [68] benchmarks in three different configurations shows (see Table 3) that migratability adds no visible overhead. Simultaneously, the latency increase for DMTCP is also minute, making it hard to distinguish all three configurations. The reason for such small difference between different configurations is the large communication overhead introduced by SoftRoCE. As a result, our experiment only shows that migratability does not add a large overhead, but does not allow to make a confident judgement regarding a potential microsecond-scale overhead. This situation is an unfortunate limitation of SoftRoCE. Considering how tiny we expect the overhead to be, even an FPGA-implementation may not be a precise reflection of a commercial RDMA NIC, because of the significant differences between an FPGA and an actual hardware implementation. We are convinced, due to the arguments given in Section 3, that MigrOS does not introduce even microsecond-scale overhead.

5.3 Migration Costs

With added support for migrating IB verbs objects, the container migration time will increase proportionally to the time required to recreate these objects. Our goal is to estimate the additional latency for migrating RDMA-enabled applications. This subsection shows the cost for migrating connections created by SoftRoCE, as well as the cost for connection creation with hardware-based IB verbs implementations.

Several IB verbs objects are required before a reliable connection (RC) can be established, see Section 2.2. Usually, an application creates a single PD, one or two CQs, multiple memory regions, and one QP per communication partner.

To measure the cost of creating individual IB verbs objects, we modified `ib_send_bw` [68] to create additional MR objects. We created one CQ, one PD, 64 QPs, and 64 1 MiB-sized MRs per run. Figure 8 shows the average time required to create

³Both versions are modified by us, because the original version (*vanilla kernel*) of the SoftRoCE driver is extremely unstable. It contained a multitude of concurrency bugs and could not be used in migration experiments. Unfortunately, fixing the race conditions required significant restructuring and resulted in a performance loss of around 20%.

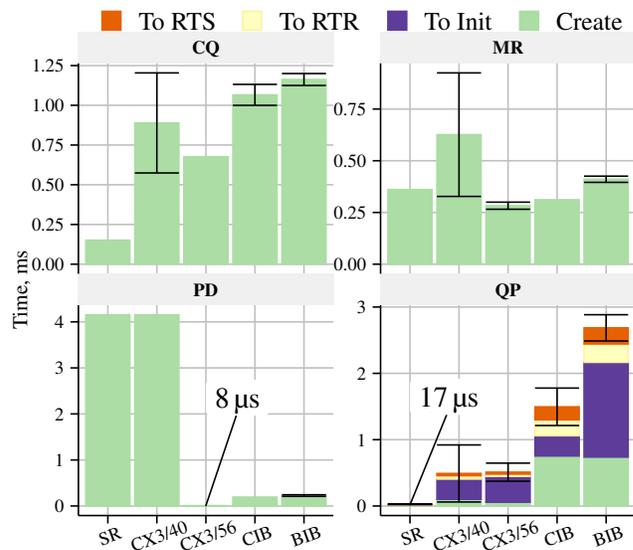


Figure 8: Object creation time for different RDMA devices (see Table 4). To send a message, a QP needs to be in the state RTS, which requires the traversal of three intermediate states (Reset, Init, RTR). Error bars show the interval of the standard deviation (σ) around the mean (μ), if $\sigma/\mu \geq 0.05$.

each object across 50 runs. Each tested NIC is represented by a bar. We draw two conclusions from this experiment. First, there is substantial variation for all operations across different NICs. Second, the time required for most operations is in the range of milliseconds.

The exact time required for migrating RDMA connections depends on two factors: the number of QPs and the total amount of memory assigned to MRs [56]. Both of these factors are application-specific and can vary greatly. Therefore, next, we show how the migration time is influenced by the application’s usage of MRs and QPs.

Figure 9 shows the MR registration time, depending on the region’s size. MR registration costs are split between the OS and the NIC: The OS pins the memory and the NIC learns about the virtual memory mapping of the registered region. SoftRoCE does not incur the “NIC-part” of the cost, so MR registration with SoftRoCE is faster than for RDMA-enabled NICs. For this experiment, we do not consider the cost of transferring the contents of the MRs during migration.

The number of QPs is the second variable influencing the migration time. Figure 10 shows the time for migrating a container running the `ib_send_bw` benchmark. This benchmark consists of two single-process containers running on two different nodes. Three seconds after communication starts, the container runtime migrates one of the containers to another node. The migration time is measured as the maximum message latency as seen by the container that did not move. The checkpoint is transferred over the same network link used by

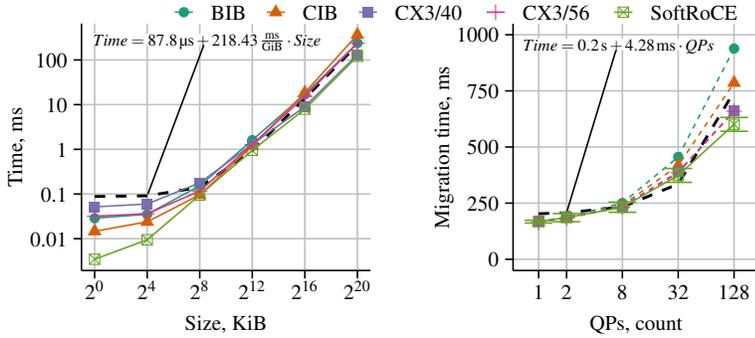


Figure 9: MR registration time depending on the region size.

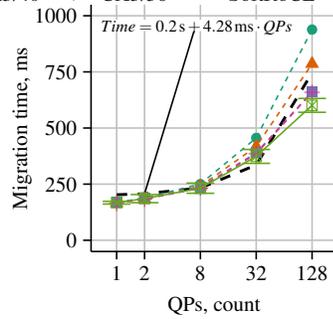


Figure 10: Migration speed with different numbers of QPs.

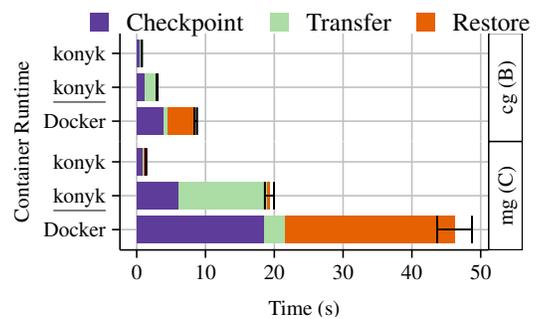


Figure 11: Migration speed of Docker and konyk with optimisations (konyk) and without (konyk)

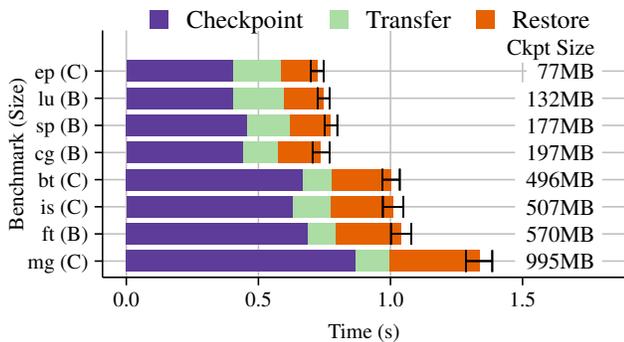


Figure 12: MPI application migration.

the benchmarks for communication. With a growing number of QPs, the benchmark consumes more memory, ranging from 8 MiB to 20 MiB. To put things into perspective, we estimated the migration time for real devices by calculating the time to recreate IB verbs object for RDMA-enabled NICs. We subtracted the time to create IB verbs objects with SoftRoCE from the measured migration time and added the time needed to create IB verbs objects with RDMA NICs (from Figure 8). We show our estimations with the dashed lines.

5.4 MPI Application Migration

For evaluating transparent live-migration of real-world applications, we chose to migrate NPB 3.4.1 [3], an MPI benchmark suite. The MPI applications run on top of Open MPI 4.0 [21], which in turn uses Open UCX 1.6.1 [77] for point-to-point communication. We configured UCX to use IB verbs communication over reliable connections (RC).

This setup corresponds to Figure 3. We containerised the applications using our self-developed runtime *konyk*, based on libcontainer [76]. Unlike Docker, our runtime facilitates faster live migration by sending the image directly to the destination node, instead of the local storage, during the checkpoint process. Additionally, *konyk* stores checkpoints in RAM, reducing migration latency even more. As any other container

runtime, *konyk* internally uses CRIU for checkpointing and restoring containers. Further description of our container runtime is out of scope of this paper.

To measure the application migration latency, we start each MPI application with four processes (*ranks*). Approximately in the middle of the application progress, one of the ranks migrates to another node. Each benchmark has a *size* (A to F) parameter. We chose the size such that all benchmarks run between 10 and 300 seconds. We excluded the “dt” benchmark, because it runs only for around a second. Figure 12 shows container migration latency and standard deviation around the mean, averaged over 20 runs of each benchmark.

We break down the migration latency into three parts: *checkpoint*, *transfer*, and *restore*. MigrOS first stops the target container and prepares the checkpoint. Almost immediately, and in parallel with checkpointing, MigrOS starts to transfer checkpoint data to the destination node. The transfer happens over the network link used by the benchmarks for communication. This overlap of checkpointing and data transfer minimises the time of exclusive data transfer. After the transfer is complete, MigrOS recovers the container at the destination node. Overall, the benchmarks experience a runtime delay proportional to the migration latency, which is proportional to the checkpoint size.

MPI applications (Figure 12) migrate slower than microbenchmarks (Figure 10), even after accounting for the checkpoint size, because of the difference in measurement methodology. For the microbenchmark, we calculate the migration time based on the maximum message latency observed by the non-migrating process. For the MPI benchmarks, we calculate the migration time from the increase in total execution time of the whole benchmark. This discrepancy indicates that the migration of parallel applications may cause a larger disruption to the application performance than the simple state transfer time can explain.

To show the interoperability of MigrOS with other container runtimes, we measured the migration costs when using Docker 19.03 (see Figure 11). We had to provide the end-to-

	Legion	Nomad	PS MPI	DMTCP	MOSIX-4	MOSIX-3	MigrOS
RDMA	✓	✓	✓	✓	✗	✗	✓
Overhead	N	N	N	Y	Y	Y	N
Runtime	✓	✓	✓				
User-OS				✓	✓		
Kernel-OS		✓				✓	✓
NIC							✓
Units	O	VM	P	P	P	P	C
Reference	[7]	[32]	[70]	[2]	[5]	[6]	Ours

Table 5: Selected checkpoint/restore systems handle either VMs, processes (P), containers (C), or application objects (O). Runtime-based systems naturally introduce no additional communication overhead for migration support.

end migration flow ourselves, because Docker features only checkpoint and restore. To our disappointment, Docker does not employ some important optimisations and requires much time to complete migration. To understand the performance difference better, we disabled some optimisations in konyk: saving checkpoints to main memory (instead of hard disk), sending checkpoints during dumping, and using an optimised way to transfer the checkpoint. All of this was not enough to match the performance of Docker. Further investigation revealed Docker unnecessarily moving checkpoint images across the file system, proving the importance of explicit live migration support within a container runtime. Nevertheless, we demonstrate the principle possibility of containerised RDMA-application migration using Docker.

6 Related Work

VMs Live migration of virtual machines (VMs) has a long usage history in cloud computing [11, 16, 31, 63, 67]. We expect live migration to become even more popular with the growth of new computing paradigms, like disaggregated and fog computing [12, 25, 65, 86]. Nevertheless, past techniques for live VM migration with RDMA NICs relied on migration-aware, paravirtualised drivers inside the VMs [32, 70]

Checkpoint/Restore Techniques Transparent live migration of processes [4, 57, 81], containers [51, 58, 62], or virtual machines [11, 16, 31, 63, 67] has long been a topic of active research. The key challenge of this technique lies in the checkpoint/restore operation. For processes and containers, this operation can be implemented at three levels: application runtime, user-level system, or kernel-level system. Table 5 compares a selection of existing checkpoint/restore systems.

Runtime-based systems expect the user application to access all external resources through the API of the runtime. This restriction resolves two important issues with resource

migratability: First, the runtime system controls exactly when the underlying resource is used and can easily stop the application from doing so to serialise the state of the resource. Second, the runtime can maintain enough information about the state of the resource to facilitate resource serialisation and deserialisation. Such interception is cheap because it happens within the application’s address space.

Almost all attempts to provide transparent live migration together with RDMA networks rely on modifications of the runtime system [2, 24, 26, 32, 41, 70]. Some runtimes operate on application-defined objects (tasks, agents, lightweight threads) for even more efficient state serialisation and deserialisation [7, 43, 87]. All runtime-based approaches bind the application to a particular runtime system.

Kernel OS-level checkpoint/restore systems [6, 28, 38, 42, 66] either do interposition at the kernel level or extract application state from the kernel’s internal data structures. Although these systems support a wider spectrum of user applications, they incur a significantly higher maintenance burden. BLCR [28] has been abandoned eventually. CRIU [13], currently the most successful OS-level tool for checkpoint/restore, keeps necessary Linux kernel modification at a minimum and does not require interposition at the user-kernel API. We describe this tool in more detail in Section 2.3.

Finally, *user OS-level* systems interpose the user-kernel API, providing the same transparency and generality as kernel-based implementations. Such systems use the `LD_PRELOAD` mechanism to intercept system calls from applications and virtualise system resources, like file descriptors, process IDs, and sockets. In version 4, MOSIX has been redesigned to work entirely at the user level [5]. DMTCP [2] is a transparent fault-tolerance tool for distributed applications with support for IB verbs. To be able to extract the state of IB verbs objects, DMTCP maintains *shadow objects*, which act as proxies between a user process and the NIC [10]. In Section 5.2, we show that maintaining these shadow objects has non-negligible runtime overhead for RDMA networks.

Furthermore, live migration may employ RDMA networks to improve the speed of the checkpoint transfer [33, 39]. These techniques allow to reduce the downtime from migration and could be combined with our technique to improve the migration time of RDMA applications.

Network Virtualisation TCP/IP network virtualisation is an essential tool for isolating distributed applications from the underlying physical network topology. Even though network virtualisation enables live migration, it introduces overhead due to additional encapsulation of network packets [64, 89]. Several new approaches try to address these performance problems [8, 64, 69, 89]. However, these approaches do not consider RDMA networks.

RDMA-network virtualisation approaches focus on implementing connection control policies in software, but do not support live container migration [29, 44, 84]. As an exception, Nomad [32] uses InfiniBand address virtualisation for VM

migration, but implements the connection migration protocol inside an application-level runtime.

MigrOS uses traditional network virtualisation for TCP/IP networks, which is not on the performance-critical path for RDMA applications. However, MigrOS avoids unnecessary interception of RDMA communication. Instead, MigrOS silently replaces addressing information during migration.

RDMA Implementations SoftRoCE [48] and SoftiWarp [83] are open-source software implementations of RoCEv2 [36] and iWarp [23] respectively. Both provide no performance advantage over socket-based communication but are compatible with their hardware counterparts and facilitate the development and testing of RDMA-based applications. We chose to base our work on SoftRoCE because RoCEv2 found wider adoption than iWarp.

There are also open-source FPGA-based implementations of network stacks. NetFPGA [90] does not support RDMA communication. StRoM [79] provides a proof-of-concept RoCEv2 implementation. However, we found it unfit to run real-world applications (for example, MPI) without further significant implementation efforts.

7 Discussion and Conclusion

MigrOS is an OS-level architecture enabling transparent live container migration without sacrificing RDMA network performance. We are convinced the architecture of MigrOS can be useful for dynamic load balancing, efficient prepared fail-over, and live software updates in cloud or HPC settings.

Hardware Modifications and Software Implementation

We believe that limited hardware changes are worthy of consideration and have already been proven feasible [14, 27, 45, 45, 61], even for RDMA protocols [34, 46, 49]. Nevertheless, propositions to modify hardware often meet criticism because they are hard to validate and evaluate for an OS designer. To overcome this difficulty, we have modified SoftRoCE. It turned out that adding only few states to the state machine and two new message types were necessary. As result, we enabled transparent live migration of containerised RDMA applications without affecting the critical path of the communication. Lesokhin et al. [46] have demonstrated that RDMA protocol changes can be achieved just through firmware updates, barring the need to replace NICs. Furthermore, our design maintains full backwards-compatibility with the existing RDMA network infrastructure at every level and can be adopted by other RDMA protocols (e.g. Infiniband and RoCEv1) verbatim.

Unreliable Datagram Communication MigrOS provides live migration for reliable communication (RC), but not for unreliable datagram (UD), because, first, every message received over UD exposes the address of its sender. When this sender migrates, its address will change and, currently, MigrOS cannot conceal the change from the receiver. Second, a

UD QP does not know where to send resume messages after migration, because it can receive messages from anywhere. Consequently, to support UD, a NIC would need to maintain an additional simple table to translate between user-visible and actual addresses. We leave this issue for future work.

Security As of today, lack of authentication and integrity control is a general problem in RDMA networks [52, 75, 80, 82]. Therefore, modern RDMA networks rely on trusted NICs and hosts. In the context of this paper, we require the host OS to ensure that pause and resume messages may only be sent by authorised hosts. If either NICs or hosts cannot be trusted, additional protocols must prevent the sending of unauthorised (e.g. by spoofing) pause and resume messages. In this regard, we do not degrade security of the RDMA network.

White-box migration Previous live migration techniques require cooperation on the application's behalf, because they see RDMA NICs as *black boxes*. To our knowledge, our work is first to consider an RDMA NIC as a *white box* for the purpose of live migration. We categorise the device state as 1. *public state*, observable through the IB verbs API, 2. *device-driver-visible state*, visible by kernel- and user-level drivers, 3. *internal state*, invisible outside the device. The device must expose its internal state to the OS at the time of migration. We hope, these findings can be useful, when implementing live migration for other devices, e.g. GPUs.

Beyond migration We believe that the pause/resume protocol can find other uses, like efficient fail-over, congestion control, or load balancing. As an example, consider MasQ [29], a virtual RDMA network with firewall capabilities. MasQ can shut down RDMA connections, but unlike TCP/IP firewalls, cannot block them temporarily. Our protocol could return control over RDMA connections to the OS and replicate TCP/IP-like behaviour to RDMA firewalls as well.

Availability github.com/TUD-OS/migros-atc-2021.

Acknowledgments

We thank our shepherd, Vasily Tarasov, and the anonymous reviewers for their helpful suggestions. The research and work presented in this paper has been supported by the German priority program 1648 “Software for Exascale Computing” via the research project FFMK [19]. This work was supported in part by the German Research Foundation (DFG) within the Collaborative Research Center HAEC and the the Center for Advancing Electronics Dresden (cfaed). The authors are grateful to the Centre for Information Services and High Performance Computing (ZIH) TU Dresden for providing its facilities. In particular, we would like to thank Dr. Ulf Markwardt and Sebastian Schrader for their support with the experimental setup. The authors thank Robert Wittig for sharing his expertise in FPGA architectures. The authors acknowledge the support from AWS Cloud Credits for Research for providing cloud computing resources.

References

- [1] Amazon Web Services, Inc. Elastic Fabric Adapter. <https://aws.amazon.com/hpc/efa/>, 2020.
- [2] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *2009 IEEE International Symposium on Parallel Distributed Processing, IPDPS*, pages 1–12, May 2009. doi:[10/d62csg](https://doi.org/10.1109/IPDPS.2009.5271111).
- [3] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatarishnan, and S.K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, September 1991. ISSN 0890-2720. doi:[10/cgsfnm](https://doi.org/10.1109/cgsfnm).
- [4] Amnon Barak and Amnon Shiloh. A distributed load-balancing policy for a multicomputer. *Software: Practice and Experience*, 15(9):901–913, September 1985. ISSN 00380644, 1097024X. doi:[10/c8r7m6](https://doi.org/10.1002/spe.390150901).
- [5] Amnon Barak and Shiloh, Amnon. The MOSIX Cluster Management System for Distributed Computing on Linux Clusters and Multi-Cluster Private Clouds. Technical report, Springer-Verlag, 2016.
- [6] Amnon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Springer-Verlag, Berlin, Heidelberg, 1993. ISBN 978-0-387-56663-4.
- [7] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, pages 1–11, Salt Lake City, UT, November 2012. IEEE. ISBN 978-1-4673-0806-9. doi:[10/gf9t42](https://doi.org/10.1109/hpcn.2012.6401422).
- [8] Adam Belay, George Prekas, Christos Kozyrakis, Ana Klimovic, Samuel Grossman, and Edouard 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, October 2014. ISBN 978-1-931971-16-4.
- [9] Brian Grant. Pod migration · Issue #3949 · kubernetes/kubernetes. <https://github.com/kubernetes/kubernetes/issues/3949>, January 2015.
- [10] Jiajun Cao, Gregory Kerr, Kapil Arya, and Gene Cooperman. Transparent checkpoint-restart over Infiniband. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing - HPDC '14*, pages 13–24, Vancouver, BC, Canada, 2014. ACM Press. ISBN 978-1-4503-2749-7. doi:[10/ggnfr4](https://doi.org/10.1145/2592048.2592063).
- [11] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2, NSDI '05*, pages 273–286. USENIX Association, May 2005. doi:[10.5555/1251203.1251223](https://doi.org/10.5555/1251203.1251223).
- [12] Connor, Patrick, Hearn, James R., Dubal, Scott P., Herdrich, Andrew J., and Sood, Kapil. Techniques to migrate a virtual machine using disaggregated computing resources, 2018.
- [13] CRIU. Checkpoint/Restore In Userspace. https://criu.org/Main_Page, 2011.
- [14] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Vivek Bhanu, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, NSDI'18, Berkeley, Calif, 2018. ISBN 978-1-931971-43-0.
- [15] Daniele De Sensi, Salvatore Di Girolamo, Kim H. McMahon, Duncan Roweth, and Torsten Hoefler. An In-Depth Analysis of the Slingshot Interconnect. *arXiv:2008.08886 [cs]*, August 2020.
- [16] Umesh Deshpande, Yang You, Danny Chan, Nilton Bila, and Kartik Gopalan. Fast Server Deprovisioning through Scatter-Gather Live Migration of Virtual Machines. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 376–383, Anchorage, AK, USA, June 2014. IEEE. ISBN 978-1-4799-5063-8. doi:[10/ggnfmm](https://doi.org/10.1109/IC3.2014.6901111).
- [17] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, March 2014.
- [18] DPDK. Data Plane Development Kit. <https://www.dpdk.org/>, 2013.
- [19] FFMK. FFMK Website. <https://ffmk.tudos.org/>, 2013.

- [20] Free Software Foundation, Inc. gcc—a Test Coverage Program. <https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>, 2021.
- [21] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241, pages 97–104, Berlin, Heidelberg, 2004. Springer. ISBN 978-3-540-30218-6. doi:[10/bxhsv5](https://doi.org/10/bxhsv5).
- [22] Peter X. Gao, Akshay Narayan, Rachit Agarwal, Sagar Karandikar, Sylvia Ratnasamy, Joao Carreira, Sangjin Han, and Scott Shenker. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI '16*, pages 249–264, Savannah, GA, USA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. doi:[10.5555/3026877.3026897](https://doi.org/10.5555/3026877.3026897).
- [23] D. Garcia, P. Culley, R. Recio, J. Hilland, and B. Metzler. A Remote Direct Memory Access Protocol Specification. <https://tools.ietf.org/html/rfc5040>, October 2007.
- [24] Rohan Garg, Gregory Price, and Gene Cooperman. MANA for MPI: MPI-Agnostic Network-Agnostic Transparent Checkpointing. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing - HPDC '19, HPDC*, pages 49–60, Phoenix, AZ, USA, 2019. ACM Press. ISBN 978-1-4503-6670-0. doi:[10/ggnd38](https://doi.org/10/ggnd38).
- [25] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with INFINISWAP. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation, NSDI '17*, page 21, Boston, MA, USA, 2017. USENIX Association. ISBN 978-1-931971-37-9. doi:[10.5555/3154630.3154683](https://doi.org/10.5555/3154630.3154683).
- [26] Wei Lin Guay, Sven-Arne Reinemo, Bjørn Dag Johnsen, Chien-Hua Yen, Tor Skeie, Olav Lysne, and Ola Tørudbakken. Early experiences with live migration of SR-IOV enabled InfiniBand. *Journal of Parallel and Distributed Computing*, 78:39–52, April 2015. ISSN 07437315. doi:[10/f68twd](https://doi.org/10/f68twd).
- [27] 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, 2015.
- [28] Paul H. Hargrove and Jason C. Duell. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series*, 46:494–499, September 2006. ISSN 1742-6588, 1742-6596. doi:[10/d33sc5](https://doi.org/10/d33sc5).
- [29] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. MasQ: RDMA for Virtual Private Cloud. 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, SIGCOMM '20*, pages 1–14, New York, NY, USA, July 2020. Association for Computing Machinery. ISBN 978-1-4503-7955-7. doi:[10/gg9rjq](https://doi.org/10/gg9rjq).
- [30] Berk Hess, Carsten Kutzner, David van der Spoel, and Erik Lindahl. GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation. *Journal of Chemical Theory and Computation*, 4(3):435–447, March 2008. ISSN 1549-9618, 1549-9626. doi:[10/b7nkp6](https://doi.org/10/b7nkp6).
- [31] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *ACM SIGOPS Operating Systems Review*, 43(3):14–26, July 2009. ISSN 0163-5980. doi:[10/ccwrpt](https://doi.org/10/ccwrpt).
- [32] Wei Huang, Jiuxing Liu, Matthew Koop, Bulent Abali, and Dhableswar Panda. Nomad: migrating OS-bypass networks in virtual machines. In *Proceedings of the 3rd international conference on Virtual execution environments - VEE '07, VEE'07*, page 158, San Diego, California, USA, 2007. ACM Press. ISBN 978-1-59593-630-1. doi:[10/frgqz4](https://doi.org/10/frgqz4).
- [33] Khaled Z. Ibrahim, Steven Hofmeyr, Costin Iancu, and Eric Roman. Optimized pre-copy live migration for memory intensive applications. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*, page 1, Seattle, Washington, 2011. ACM Press. ISBN 978-1-4503-0771-0. doi:[10/bsgrkf](https://doi.org/10/bsgrkf).
- [34] InfiniBand Trade Association. *Supplement to InfiniBand Architecture Specification: XRC*. InfiniBand Trade Association, February 2009.
- [35] InfiniBand Trade Association. *Supplement to InfiniBand Architecture Specification: RoCE*, volume 1. InfiniBand Trade Association, 1.2.1 edition, 2010.
- [36] InfiniBand Trade Association. *Supplement to InfiniBand Architecture Specification: RoCEv2*. InfiniBand Trade Association, September 2014.
- [37] InfiniBand Trade Association. *InfiniBand Architecture Specification*, volume 1. InfiniBand Trade Association, 1.3 edition, March 2015.

- [38] Jake Edge. Checkpoint/restart tries to head towards the mainline. <https://lwn.net/Articles/320508/>, February 2009.
- [39] Joel Nider and Mike Rapoport. News from academia: FatELF, RDMA and CRIU. In *Linux Plumbers Conference*, Vancouver, BC, Canada, November 2018.
- [40] Jonathan Corbet. TCP connection repair. <https://lwn.net/Articles/495304/>, May 2012.
- [41] J. Jose, Mingzhe Li, Xiaoyi Lu, K. C. Kandalla, M. D. Arnold, and D. K. Panda. SR-IOV Support for Virtualization on InfiniBand Clusters: Early Experience. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, Delft, May 2013. IEEE. ISBN 978-0-7695-4996-5. doi:10/ggm53b.
- [42] Asim Kadav and Michael M. Swift. Live migration of direct-access devices. *ACM SIGOPS Operating Systems Review*, 43(3):95, July 2009. ISSN 01635980. doi:10/b9j36z.
- [43] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: a portable concurrent object oriented system based on C++. *ACM SIGPLAN Notices*, 28(10):91–108, October 1993. ISSN 0362-1340. doi:10/cgnqf7.
- [44] 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 *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI, pages 113–125, Boston, MA, USA, 2019. ISBN 978-1-931971-49-2. doi:10.5555/3323234.3323245.
- [45] Alec Kochevar-Cureton, Somesh Chaturmohta, Norman Lam, Sambhrama Mundkur, and Daniel Firestone. Remote direct memory access in computing systems, October 2019.
- [46] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafirir. Page Fault Support for Network Controllers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 449–466, Xi’an China, April 2017. ACM. ISBN 978-1-4503-4465-4. doi:10/ghm5x7.
- [47] Linux Containers - LXC - Introduction. Linux Containers (LXC). <https://linuxcontainers.org/lxc/>, 2008.
- [48] Liran Liss. The Linux SoftRoCE Driver, March 2017.
- [49] Liran Liss. On Demand Paging for User-level Networking, 2013.
- [50] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High performance RDMA-based MPI implementation over InfiniBand. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS ’03, pages 295–304, San Francisco, CA, USA, June 2003. Association for Computing Machinery. ISBN 978-1-58113-733-0. doi:10/c4knj6.
- [51] Lele Ma, Shanhe Yi, and Qun Li. Efficient service hand-off across edge servers via Docker container migration. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC ’17, pages 1–13, San Jose, California, 2017. ACM Press. ISBN 978-1-4503-5087-7. doi:10/gf9x9r.
- [52] Manhee Lee, Eun Jung Kim, and M. Yousif. Security enhancement in InfiniBand architecture. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 10 pp.–, April 2005. doi:10/c4xpq5.
- [53] Mellanox. Messaging Accelerator (VMA) Documentation. Technical report, Mellanox, April 2019.
- [54] Mellanox Technologies. RDMA Aware Networks Programming User Manual. Technical Report 1.7, Mellanox Technologies, 2015.
- [55] Microsoft. High performance computing VM sizes. <https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-hpc>, August 2020.
- [56] Frank Mietke, Robert Rex, Robert Baumgartl, Torsten Mehlan, Torsten Hoeffler, and Wolfgang Rehm. Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner, editors, *EuroPar 2006 Parallel Processing*, volume 4128, pages 124–133. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-37783-2 978-3-540-37784-9. doi:10.1007/11823285_13.
- [57] Dejan Milojević, Frederick Douglass, and Richard Wheeler. *Mobility: processes, computers, and agents*. ACM Press/Addison-Wesley Publishing Co., USA, 1999. ISBN 978-0-201-37928-0.
- [58] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Linux Symposium*, Ottawa, 2008.

- [59] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, 2013. ISBN 978-1-931971-01-0.
- [60] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18*, pages 313–326, New York, NY, USA, August 2018. Association for Computing Machinery. ISBN 978-1-4503-5567-4. doi:[10/gghf6mq](https://doi.org/10/gghf6mq).
- [61] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI '20*, pages 77–92, Santa Clara, CA, USA, February 2020. USENIX Association. ISBN 978-1-939133-13-7.
- [62] Shripad Nadgowda, Sahil Suneja, Nilton Bila, and Canturk Isci. Voyager: Complete Container State Migration. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2137–2142, Atlanta, GA, USA, June 2017. IEEE. ISBN 978-1-5386-1792-2. doi:[10/ggnhq5](https://doi.org/10/ggnhq5).
- [63] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference, ATC '05*, pages 391–394, Anaheim, CA, USA, April 2005. USENIX Association. doi:[10.5555/1247360.1247385](https://doi.org/10.5555/1247360.1247385).
- [64] Zhixiong Niu, Hong Xu, Peng Cheng, Yongqiang Xiong, Tao Wang, Dongsu Han, and Keith Winstein. NetKernel: Making Network Stack Part of the Virtualized Infrastructure. *arXiv:1903.07119 [cs]*, March 2019.
- [65] Opeyemi Osanaiye, Shuo Chen, Zheng Yan, Rongxing Lu, Kim-Kwang Raymond Choo, and Mqhele Dlodlo. From Cloud to Fog Computing: A Review and a Conceptual Live VM Migration Framework. *IEEE Access*, 5:8284–8300, 2017. ISSN 2169-3536. doi:[10/ggnfkt](https://doi.org/10/ggnfkt).
- [66] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of Zap: a system for migrating computing environments. *ACM SIGOPS Operating Systems Review*, 36(SI):361–376, December 2003. ISSN 0163-5980. doi:[10/fbg7vq](https://doi.org/10/fbg7vq).
- [67] Zhenhao Pan, Yaozu Dong, Yu Chen, Lei Zhang, and Zhijiao Zhang. CompSC: live migration with pass-through devices. *ACM SIGPLAN Notices*, 47(7):109, September 2012. ISSN 03621340. doi:[10/f3887q](https://doi.org/10/f3887q).
- [68] perftest. perftest. <https://github.com/linux-rdma/perftest>, April 2020.
- [69] 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. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, 2014, OSDI '14*, 2014.
- [70] S. Pickartz, C. Clauss, S. Lankes, S. Krempel, T. Moschny, and A. Monti. Non-intrusive Migration of MPI Processes in OS-Bypass Networks. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1728–1735, May 2016. doi:[10/ggscxh](https://doi.org/10/ggscxh).
- [71] podman.io. Podman: daemonless container engine. <https://podman.io/>, 2018.
- [72] Marius Poke and Torsten Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing - HPDC '15*, pages 107–118, Portland, Oregon, USA, 2015. ACM Press. ISBN 978-1-4503-3550-8. doi:[10/ggm3sf](https://doi.org/10/ggm3sf).
- [73] proc - process information pseudo-filesystem. proc(5) - Linux manual page. <http://man7.org/linux/man-pages/man5/proc.5.html>, 2020.
- [74] ptrace - process trace. ptrace(2) - Linux manual page. <http://man7.org/linux/man-pages/man2/ptrace.2.html>, 2020.
- [75] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMARK: Bypassing RDMA Security Mechanisms. In *30th USENIX Security Symposium*, page 16, Vancouver, B.C., 2021.
- [76] runc. runc: CLI tool for spawning and running containers according to the OCI specification. <https://github.com/opencontainers/runc>, 2020.
- [77] Pavel Shamis, Manjunath Gorentla Venkata, M. Graham Lopez, Matthew B. Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L. Graham, Liran Liss, Yiftah Shahar, Sreeram Potluri, Davide Rossetti, Donald Becker, Duncan Poole, Christopher Lamb, Sameer Kumar, Craig Stunkel, George Bosilca, and Aurelien Bouteiller. UCX: An Open Source Framework for HPC Network APIs and Beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, HotI*, pages 40–43, August 2015. doi:[10/ggm8k](https://doi.org/10/ggm8k).

- [78] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and Concurrent RDF Queries with RDMA-based Distributed Graph Exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI '16*, page 17, 2016.
- [79] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. StRoM: Smart Remote Memory. In *the Fifteenth European Conference on Computer Systems, EuroSys '20*, page 16, Heraklion, Greece, 2020. Association for Computing Machinery. doi:[10.1145/3342195.3387519](https://doi.org/10.1145/3342195.3387519).
- [80] Anna Kornfeld Simpson, Adriana Szekeres, Jacob Nelson, and Irene Zhang. Securing RDMA for High-Performance Datacenter Storage Systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 20*, page 8. USENIX Association, 2020.
- [81] Jonathan M. Smith. A survey of process migration mechanisms. *ACM SIGOPS Operating Systems Review*, 22(3):28–40, July 1988. ISSN 0163-5980. doi:[10/bjp787](https://doi.org/10/bjp787).
- [82] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefer. sRDMA – Efficient NIC-based Authentication and Encryption for Remote Direct Memory Access. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 691–704, 2020. ISBN 978-1-939133-14-4.
- [83] Animesh Trivedi, Bernard Metzler, and Patrick Stuedi. A case for RDMA in clouds: turning supercomputer networking into commodity. In *Proceedings of the Second Asia-Pacific Workshop on Systems - APSys '11*, page 1, Shanghai, China, 2011. ACM Press. ISBN 978-1-4503-1179-3. doi:[10/fzv576](https://doi.org/10/fzv576).
- [84] Shin-Yeh Tsai and Yiying Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles - SOSPP '17*, pages 306–324, Shanghai, China, 2017. ACM Press. ISBN 978-1-4503-5085-3. doi:[10/ggscxn](https://doi.org/10/ggscxn).
- [85] Dongyang Wang, Binzhang Fu, Gang Lu, Kun Tan, and Bei Hua. vSocket: virtual socket interface for RDMA in public clouds. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE*, pages 179–192, Providence, RI, USA, 2019. ACM Press. ISBN 978-1-4503-6020-3. doi:[10/ggscxg](https://doi.org/10/ggscxg).
- [86] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. In *Proceedings of the Fourteenth EuroSys Conference 2019, EuroSys '19*, pages 1–16, Dresden, Germany, March 2019. Association for Computing Machinery. ISBN 978-1-4503-6281-8. doi:[10/ggnq76](https://doi.org/10/ggnq76).
- [87] David Wong, Noemi Paciorek, and Dana Moore. Java-based mobile agents. *Communications of the ACM*, 42(3):92–ff., March 1999. ISSN 0001-0782. doi:[10/btg3k7](https://doi.org/10/btg3k7).
- [88] Yibo Zhu, Ming Zhang, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, and Mohamad Haj Yahia. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM*, pages 523–536, London, UK, 2015. ACM. ISBN 978-1-4503-3542-3. doi:[10.1145/2785956.2787484](https://doi.org/10.1145/2785956.2787484).
- [89] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation, NSDI*, Boston, MA, USA, February 2019. USENIX Association. ISBN 978-1-931971-49-2.
- [90] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 34(5):32–41, September 2014. ISSN 1937-4143. doi:[10/gg8qsd](https://doi.org/10/gg8qsd).

Prediction-Based Power Oversubscription in Cloud Platforms

Alok Kumbhare, Reza Azimi, Ioannis Manousakis, Anand Bonde, Felipe Frujeri, Nithish Mahalingam, Pulkit A. Misra, Seyyed Ahmad Javadi, Bianca Schroeder, Marcus Fontoura, and Ricardo Bianchini *

Microsoft Research and Microsoft Azure

Abstract

Prior work has used power capping to shave rare power peaks and add more servers to a datacenter, thereby oversubscribing its resources and lowering capital costs. This works well when the workloads and their server placements are known. Unfortunately, these factors are unknown in public clouds, forcing providers to limit the oversubscription and thus the potential performance loss from power capping. In this paper, we argue that providers can use predictions of workload performance criticality and virtual machine (VM) resource utilization to increase oversubscription. This poses many challenges, such as identifying the performance-critical workloads from opaque VMs, creating support for criticality-aware power management, and increasing oversubscription while limiting the impact of capping. We address these challenges for the hardware and software of Microsoft Azure. The results show that we enable a $2\times$ increase in oversubscription with minimum impact to critical workloads. We describe lessons from deploying our work in production.

1 Introduction

Motivation. Large Internet companies continue building datacenters to meet the increasing demand for their services. Each datacenter costs hundreds of millions of dollars to build. Power plays a key role in datacenter design, build out, IT capacity deployment, and physical infrastructure cost.

The power delivery infrastructure forms a hierarchy of devices that supply power to different subsets of the deployed IT capacity at the bottom level. Each device includes a circuit breaker to prevent damage to the IT infrastructure in the event of a power overdraw. When a breaker trips, the hardware downstream loses power, causing a partial blackout.

To avoid tripping breakers, designers conservatively provision power for each server based on either its maximum nameplate power or its peak draw while running a power-hungry benchmark, such as SPEC Power [25]. The maximum number of servers is then the available power (or breaker limit) divided by the per-server provisioned value. This provisioning leads to massive power under-utilization. As the IT demand increases, it also requires building new datacenters even when there are available resources (space, cooling, networking) in existing ones, thus incurring huge unnecessary capital costs.

To improve efficiency and avoid these costs, prior work has proposed combining power capping and oversubscription [12, 43]. The idea is to leverage actual server utilization

and statistical multiplexing across workloads to oversubscribe the delivery infrastructure by *adding more servers to the datacenter*, while ensuring that the power draw remains below the breakers' limits. This is achieved by continuously monitoring the power draw at each level and using power capping (via CPU voltage/frequency and memory bandwidth throttling), when necessary. As throttling impacts performance, these approaches carefully define which workloads can be throttled and by how much. For example, Facebook's Dynamo relies on predefined workload priority groups, and throttles each server based on the priority of the workload it runs [43].

This oversubscription approach works well when workloads and their server placements are known. However, *public cloud platforms violate these assumptions*. First, each server runs many VMs, each with its workload, performance, and power characteristics. Hence, throttling the entire server would impact performance-critical (e.g., interactive services) and non-critical (e.g., batch) workloads alike. Second, VMs dynamically arrive and depart from each server, producing varying mixes of characteristics and preventing predefined server groupings or priorities. Third, each VM must be treated as a black box, as customers are often reluctant to accept deep inspection of their VMs. Thus, the platform does not know which VMs are performance-critical and which ones are not. For these reasons, *oversubscription in public clouds has been limited so that performance is never impacted*.

Our work. In this paper, we argue that cloud providers can increase oversubscription substantially by carefully scheduling VMs and managing power, based on predictions of workload performance criticality and VM CPU utilization. Our insight is that there are many non-critical workloads (e.g., batch jobs) that can tolerate a slightly higher rate of capping events and/or deeper throttling; the capping of performance-critical workloads must be controlled more tightly. Using predictions to identify these workloads and place them carefully across the datacenter provides the power slack and criticality-awareness needed to increase oversubscription.

Accurately predicting workload criticality from outside opaque VMs is itself a challenge. Prior work [9] associated a diurnal utilization pattern with user interactivity and the critical need for high performance. Here, we present an accurate and robust pattern-matching algorithm to infer criticality, and a machine learning (ML) model that uses the algorithm's output during training. We also propose a model for predicting the 95th-percentile CPU utilization over a VM's lifetime.

*Azimi, Javadi and Schroeder were at Microsoft Research for this work.

With these predictions, we increase the power slack in the datacenter by balancing the expected power draw and our ability to lower it via throttling when a power budget is exceeded (causing a capping event). We accomplish this with a criticality- and utilization-aware VM placement policy. When events occur, we must cap power intelligently as well. So, we propose a system that protects performance-critical VMs from throttling when capping a server's power draw.

Using the above contributions and the history of power draws, we devise a new strategy for selecting the amount of oversubscription. The strategy limits the impact of capping on the two VM types to predefined acceptable values, thereby enabling significant but controlled increases in oversubscription. *Providers that prefer to treat all external (i.e., third-party) VMs the same can simply assume them all to be critical, and classify only the internal (i.e., first-party) VMs into the two types at the cost of a lower increase in oversubscription.*

We implement our work for the infrastructure of Microsoft Azure. The evaluation shows that our criticality algorithm and ML models are very accurate. We also show that our system and policy lower the performance impact of a capping event, while our policy produces fewer events. Overall, we can increase oversubscription by $2\times$ (from 6 to 12%), compared to the state-of-the-art approach. This increase would save \$75.5M in capital costs from *each* datacenter site (128MW). Assuming that all external VMs are performance-critical would lower the savings to a still significant \$28.1M. We have also started deploying our work in production in Azure and mention some lessons in Section 5.

Related work. The prior work on power capping [16, 19, 26, 27, 29–31, 34, 36, 45] and oversubscription [12, 14, 18, 28, 38, 39, 42, 43] produced major advances in server and datacenter power management. Unfortunately, it falls short for real public clouds. For example, it has capped server power using inputs that are typically not available in the public cloud, such as application-level metrics or operator annotations. Moreover, it has employed reactive and expensive VM migration in clusters, instead of leveraging predictions for capping and capping-aware scheduling. Prior oversubscription works have focused on non-cloud datacenters and full-server capping when workloads and their priorities are known.

Summary. We make the following main contributions:

1. An algorithm and ML model for predicting performance criticality, and a model for predicting VM utilization.
2. A VM placement policy that uses these predictions to minimize the number of capping events and their impact.
3. A per-VM power capping system that uses predictions of criticality to protect certain VMs.
4. A strategy that leverages the contributions above to increase the amount of oversubscription.
5. Implementation and results for Azure's infrastructure, showing large potential cost savings.
6. Lessons from production deployment of our work.

Though we build upon Azure's infrastructure, our concep-

tual contributions (e.g., predicting criticality; using predictions in VM placement, power budget enforcement, and oversubscription) apply directly to any cloud platform. Similarly, although we focus on VMs, our contributions also apply to containers running on bare-metal servers. As providers want to maximize the use of their servers via multi-tenancy, each container typically runs on a lightweight VM for security isolation [3, 32]. We can treat these VMs like any other. For scenarios where isolation between containers is not required, we can treat the whole server as a single workload or adapt our software to treat containers as we treat VMs.

2 Background and context

2.1 Typical power delivery, server deployment

At the top of the power delivery hierarchy, the electrical grid provides power to a sub-station that is backed up by a generator. An Uninterruptible Power Supply (UPS) unit provides battery backup while the generator is starting up. The UPS feeds one power distribution unit (PDU) per row of servers. Each row PDU supplies power to several rack PDUs, each of which feeds a few server chassis. Each chassis contains a few power supplies (PSUs) and dozens of blade servers.¹ A PDU trips its circuit breaker when the power draw exceeds the rated value (budget) for the unit, causing a power outage.

Designers deploy servers so that breakers never trip, leading to wasted resources (chiefly space, cooling, and networking). Combining hierarchical power capping and oversubscription enables more capacity to be deployed and better utilizes resources [12, 43]. For example, if designers find that the historical per-row power draw is consistently lower than the row PDU budget, they can “borrow” power from each row to add more rows (until they run out of row space) under the UPS budget. The extra rows oversubscribe the power at the UPS level. The servers downstream from the oversubscribed PDUs/UPS must be power-capped, whenever they are about to draw power that has already been borrowed.

2.2 Azure's existing capping mechanisms

For clarity and ease of experimentation, in this paper we explore power budget enforcement at the chassis level.

Azure sets power budgets for each chassis, where each blade in the chassis is allocated its even share of the chassis budget; uneven blade budgets are infeasible due to the overhead of dynamically reapportioning and reinstalling budgets. No capping takes place under normal operation, i.e. each blade is free to draw more power than its even share, as long as the total chassis draw is below the budget. The PSUs alert the board management chip (BMC) on blades directly when the chassis budget is about to be exceeded. Upon an alert, the blade power must be brought below its even-share cap. The BMC splits the cap evenly across its sockets and uses Intel's Running Average Power Limit (RAPL) [10] to lower

¹We refer to “blades” and “servers” interchangeably throughout the paper.

the blade power. RAPL throttles the entire socket (slowing down *all cores equally*) and memory using a feedback loop until the cap is respected. Typically, RAPL brings the power below the cap in less than 2 seconds. This reaction time is sufficient for power safety because leaf-level PDUs have a high (e.g., 7x) overload tolerance over 2 seconds, according to their breaker-trip curves [40]. Such overdraws are impractical even with aggressive oversubscription.

2.3 Azure’s existing VM scheduler

A cloud platform first routes an arriving VM to a server cluster. Within each cluster, a VM scheduler is responsible for placing the VM on a server. The scheduler uses heuristics to tightly pack VMs, considering the incoming VM’s multiple resource requirements and each server’s available resources.

Azure’s scheduler [17] implements its heuristics as two sets of rules. It first applies *constraint* rules (e.g., does the server have enough resources for the VM?) to filter invalid servers, and then applies *preference* rules, each of which orders the candidate servers based on a preferred metric (e.g., a packing score derived from available resources). It then weights each candidate based on its order on the preference list for each rule and the rule’s pre-defined weight. Finally, it picks a server with the highest aggregate weight for allocation. No rules currently consider power draws or capping.

2.4 Azure’s existing ML system

To integrate predictions into VM scheduling in practice, we use Resource Central [6], the existing ML and prediction-serving system in Azure. The system provides a REST service for clients (e.g., the VM scheduler) to query for predictions. It can receive input features (e.g., user, VM size, guest OS) that are known at deployment time from the scheduler, execute one of our ML models (criticality or CPU utilization), and respond with the prediction and a confidence score. Model training is done in the background, e.g. once a day.

3 Prediction-based oversubscription

Cloud providers provision servers conservatively, and have full-server capping mechanisms and capping-oblivious VM schedulers. We propose to provision servers more aggressively by making the infrastructure smarter and finer-grained via VM behavior predictions. Key challenges include having to create or adapt certain components (e.g., scheduler, chassis manager) to the predictions, while controlling the tradeoff between oversubscription and performance tightly.

Next, we overview our design and detail its main components. Then, we describe our strategy for provisioning servers to balance cost savings and performance impact.

3.1 Overview

Figure 1 overviews our system and its operation, showing the existing and new/modified components in different colors. We modify the VM scheduler to use predictions of VM performance criticality and resource utilization. We implement

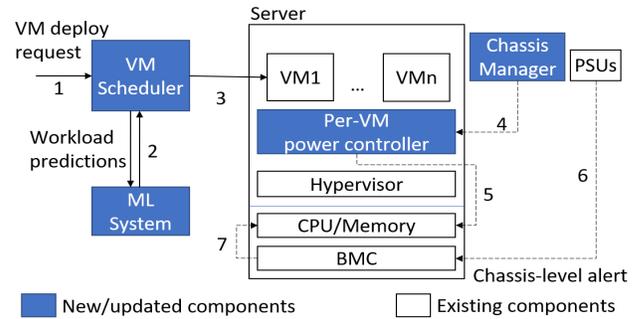


Figure 1: System overview.

our ML models so they can be managed and served by Resource Central. We modify the chassis manager to query the chassis power draw and interact with our new per-VM power controller. The controller manages its server’s power draw during a capping event.

A request to deploy a set of VMs arrives at our VM scheduler (arrow #1). The scheduler then queries the ML system for predictions of workload performance criticality and resource utilization (#2). Using these predictions, it decides on which servers to place the VMs (#3). After selecting each VM’s placement, the scheduler tags the VM with its predicted workload type and instructs the destination server to create it. Each chassis manager polls its local PSUs to determine whether the power draw for the chassis crosses a threshold just below the chassis budget. (This threshold enables the controller to perform per-VM capping and hopefully avoid needing full-server RAPL.) When this is the case, the manager alerts the controller of each server in the chassis (#4).

Upon receiving the alert, our controller at each server manages the server’s (even) share of the chassis budget across the local VMs based on their workload types. It does this by first throttling the CPU cores used by non-performance-critical VMs (#5). Throttling these VMs may be enough to keep the power below the chassis budget, and protect the performance-critical VMs. If it is not enough, the PSUs alert the servers’ BMCs (#6), which will then use RAPL as a last resort to lower the chassis power below the budget (#7).

Limiting impact on non-critical VMs. Though we protect critical VMs from throttling, we also limit the performance impact on non-critical VMs in three ways. First, for long-term server provisioning, our oversubscription strategy carefully selects chassis budgets to limit the number of capping events and their severity to predefined acceptable values (e.g., no more than 1% events for non-critical VMs, each lowering the core frequency to no less than 75% of the maximum). Second, for medium-term management, our scheduler places VMs seeking to minimize the number of events and their severity. Finally, in the shortest term, our per-VM controller increases the core frequency of non-critical VMs as soon as possible, to keep a server’s power close to (but below) the limit during a capping event (Section 3.4).

Treating external VMs as performance-critical. Providers

who prefer to treat all paying customers the same can easily do so by assuming that all external VMs are performance-critical; only internal VMs (e.g., running the provider’s own managed services) would be classified into the two criticality types. We explore this assumption in Section 4.6.

3.2 Predicting VM criticality and utilization

Our approach depends on predicting VM performance criticality and utilization at arrival time, i.e. just before the VMs are deployed. We train supervised ML models to produce these predictions based on historical VM arrival data and telemetry that was collected after those VMs were deployed. Since VMs are black boxes, our telemetry consists of CPU utilization data only, as deep inspection is not an option.

Inferring criticality. Predicting criticality requires a method to determine the VM *labels*, i.e. whether the workload of each VM is performance-critical or not, before we can train a model. As in prior work [9], we consider a workload critical if it is user-facing, i.e. a human is interacting with the workload (e.g., front-end webservers, backend databases), and less critical otherwise (e.g., batch, development and testing workloads). As user-facing workloads exhibit utilization patterns that repeat daily (e.g., high during the day, low at night), the problem reduces to identifying VMs whose time series of CPU utilizations exhibit 24-hour periods [9].

Although some background VMs may exhibit 24-hour periods, this is not a problem as we seek to be conservative (i.e., it is fine to classify a non-user-facing workload as user-facing, but not vice-versa). Moreover, some daily batch jobs have strict deadlines, so classifying them as user-facing correctly reflects their needs. Importantly, focusing on the CPU utilization signal works well even when the CPU is not the dominant resource, as the CPU is always a good proxy for periodicity (e.g., network-bound interactive workloads exhibit more CPU activity during the day than at night). This is true even for workloads that use load-based auto-scaling [2, 33], as auto-scaling impacts VM utilization but does not create or destroy periodicity, e.g. a VM deployment that receives diurnal load will show periodicity whether auto-scaling is enabled or not.

We considered but discarded other approaches for inferring whether a VM’s workload is user-facing. For example, observing whether a VM exchanges messages does not work because many non-user-facing workloads communicate externally (e.g., to bring data in for batch processing).

Identifying periodicity. There are statistical methods for identifying periods in time series, such as FFT or the auto-correlation function (ACF). For example, [9] assumes a workload is user-facing if the FFT indicates a 24-hour period. *We evaluated ACF and FFT methods on 840 workloads.* Surprisingly, we find that both methods lead to frequent misclassifications. We identify three culprits:

1. The diurnal patterns in user-facing workloads often have significant noise and interruptions. For example, we observe user-facing workloads with clear 24-hour periods for many

days, interrupted by a period of constant or random load, causing them to be mis-classified as non-user-facing.

2. The diurnal patterns often exhibit increasing/decreasing trends (e.g., the workload becomes more popular over time), and varying magnitudes of peaks/valleys across days. These effects cause some user-facing workloads to be mis-classified.

3. There are many machine-generated workloads with periods of 1 hour, 4 hours or other divisors of 24 hours, which therefore also have 24-hour periods, leading to machine-generated workloads that are mis-classified as user-facing.

Part of the problem is that ACF and FFT are very general tools with different goals, e.g. decomposing a signal for compact representation and capturing general correlations, not solutions for our specific problem of 24-hour periods.

Criticality algorithm. Thus, we devise a new algorithm that is more robust and targeted at our specific problem. Our idea is to extract from a VM’s utilization time series a *template* for a typical 24-hour period and then check how well this template captures most days in the series. We design the template extraction and comparison to be robust to noise and interruptions to deal with issue #1 above. We pre-process the data using methods from time series analysis to address #2. To deal with #3, we extract templates for shorter periods (8 and 12 hours) and ensure that the 24-hour template is the best fit. These periods subsume the other short periods.

More precisely, the input to our pattern-matching algorithm is the average CPU utilization for each 30-minute interval over 5 weekdays; this duration is long enough to unearth any periodicity in a VM’s CPU utilization signal. (Shorter workloads cannot be classified and should be conservatively assumed user-facing.) For each utilization time series, the algorithm does the following:

1. It de-trends and normalizes the time series, so that all days exhibit utilizations within the same rough range. De-trending scales each utilization based on the mean of the previous 24 hours, whereas normalization divides each utilization by the standard deviation of the whole time series.

2. It extracts the 24-hour template by identifying, for each time of the day (in 30-minute chunks), its “typical” utilization computed as the median of all utilizations in the pre-processed series that were reported at this time of the day.

3. It overlays the template over the pre-processed series for each day and computes the average deviation for each utilization, after excluding the 20% largest deviations.

4. It repeats steps 2 and 3 to compute average deviations for 8-hour and 12-hour templates, and then computes two scores: 24-hour average deviation divided by 8-hour average deviation (called `Compare8`), and 24-hour average deviation divided by 12-hour average deviation (called `Compare12`). If the scores are close to 0, the workload is likely to be user-facing. Ultimately, it classifies a time series as user-facing, if its `Compare8` value is lower than a threshold (Section 4.2).

Criticality prediction. The algorithm above produces labels that we use to train an ML model to classify arriving VMs as

user-facing or non-user-facing. Specifically, we train a Random Forest using the labels and many features (pertaining to the arriving VM and its cloud subscription) available at arrival time: the percentage of user-facing VMs in the subscription, the percentage of VMs that lived at least 7 days in the subscription, the total number of VMs in the subscription, the percentage of VMs in each CPU utilization bucket, the averages of the VMs' average and 95th-percentile CPU utilizations in the subscription, the arriving VM's number of cores and memory size, and the arriving VM's type.

Utilization prediction. For utilization predictions, we train a two-stage model to predict 95th-percentile VM CPU utilization based on labels produced by previous VM executions (actual 95th-percentile utilizations over the VMs' lifetimes) and the same VM features we use in the criticality model. Since predicting utilization exactly is hard, our model predicts it into 4 buckets: 0%-25%, 26%-50%, and so on. The first stage is a Random Forest that predicts whether or not the 95th-percentile utilization is above 50%. In the second stage, we have a Random Forest for buckets 1-2 and another for buckets 3-4. We train these latter forests with just the VMs we can predict with high-confidence ($\geq 60\%$) in the first stage.

We experimented with single-stage models, but they did not produce accurate predictions with enough confidence.

Model training and inference. Resource Central trains our prediction models in the background once a day. It also monitors prediction accuracy; we do not find significant improvements from more frequent training. The models exhibit sub-millisecond prediction latency, which is a small fraction of VM creation times [1].

3.3 Modified VM scheduler

Our ability to increase oversubscription and the efficacy of the per-VM power controller depend on the placement of VMs in each cluster. Better placements have a balanced distribution of power draws across the different chassis to reduce the number of capping events (Goal #1); and a balanced distribution of cap-able power (drawn by non-user-facing VM cores) across servers, so the controller can lower the power during an event without affecting critical VMs (Goal #2). The scheduler must remain effective at packing VMs while minimizing the number of deployment failures (Goal #3).

Given these goals, we modify Azure's VM scheduler to become criticality- and utilization-aware, using predictions at VM arrival time. Our policy is a preference rule that sorts the feasible servers based on a "score". Each server's score considers the predicted 95th-percentile CPU utilization of the VMs already placed in the same chassis (targets Goal #1), and the predicted criticality and 95th-percentile CPU utilization of the VMs already placed on the same server (targets Goal #2). The policy only considers CPU utilization because the CPUs are the dominant source of dynamic power in Azure's servers. Moreover, throttling the CPUs typically reduces the power of other resources (e.g., memory, storage) because of

Algorithm 1 Criticality- & utilization-aware VM placement

```

1: function SORTCANDIDATES( $V, \zeta$ )
    $\triangleright V$ : VM to be placed,  $\zeta$ : list of candidate servers
2:    $\omega \leftarrow V^{PredictedWorkloadType}$ 
3:   for  $c_i$  in  $\zeta$  do
4:      $\kappa_i \leftarrow SCORECHASSIS(c_i.Chassis)$ 
5:      $\eta_i \leftarrow SCORESERVER(\omega, c_i)$ 
6:      $c_i.score \leftarrow \alpha \times \kappa_i + (1 - \alpha) \times \eta_i$ 
7:   return  $\zeta.SORTDESC(c_i.score)$ 
8: function SCORECHASSIS( $C$ )
9:   for  $n_i$  in  $C.Servers$  do
10:    for  $v_j$  in  $n_i^{VMs}$  do
11:       $\rho^{Peak} \leftarrow \rho^{Peak} + v_j^{PredictedP95Util} \times v_j^{cores}$ 
12:       $\rho^{Max} \leftarrow \rho^{Max} + n_i^{cores}$ 
13:    return  $1 - \left[ \frac{\rho^{Peak}}{\rho^{Max}} \right]$ 
14: function SCORESERVER( $\omega, N$ )
15:   for  $v_i$  in  $N^{UF\_VMs}$  do
16:      $\gamma^{UF} \leftarrow \gamma^{UF} + v_i^{PredictedP95Util} \times v_i^{cores}$ 
17:   for  $v_i$  in  $N^{NUF\_VMs}$  do
18:      $\gamma^{NUF} \leftarrow \gamma^{NUF} + v_i^{PredictedP95Util} \times v_i^{cores}$ 
19:   if  $\omega = UF$  then
20:     return  $\frac{1}{2} \times \left( 1 + \frac{\gamma^{NUF} - \gamma^{UF}}{N^{cores}} \right)$ 
21:   else
22:     return  $\frac{1}{2} \times \left( 1 + \frac{\gamma^{UF} - \gamma^{NUF}}{N^{cores}} \right)$ 

```

the reduced number of accesses per second coming from the CPUs. As Section 4.5 shows, our policy does not degrade the packing of VMs onto servers, nor does it increase the percentage of VM deployment failures (achieves Goal #3).

Algorithm 1 shows our rule (*SortCandidates*) and two supporting routines. We show the predictions with the *PredictedWorkloadType* and *PredictedP95Util* superscripts. The rule ultimately computes the score for each candidate server (line #6). The higher the score, the more preferable the server. The score is a function of how preferable the server (line #5) and its chassis (line #4) are for the VM to be placed. Both server and chassis intermediate scores range from 0 to 1. We weight the intermediate scores to give them differentiated importance. We select the best value for the α weight in Section 4.5.

Function *ScoreChassis* computes the chassis score for a candidate server by conservatively estimating its aggregate chassis CPU utilization, i.e. assuming all VMs scheduled to the chassis are at their individual 95th-percentile utilization at the same time. This value is the sum of the predicted 95th-percentile utilizations for the VMs scheduled to the chassis, divided by the maximum core utilization (#cores in chassis $\times 100\%$). This ratio is proportional to utilization. We subtract it from 1, so that chassis with low utilization get higher values and are preferred (line #13).

Function *ScoreServer* scores a candidate server differently depending on the type of VM that is being deployed. First, it sums up the predicted 95th-percentile utilizations of the user-facing VMs (lines #15-16) and non-user-facing VMs (lines #17-18) independently. When a user-facing VM is being deployed, we compute how much more utilized the non-user-

facing VMs on the server are than the user-facing ones. We do the reverse for a non-user-facing VM. The reversal is the key to balancing the cap-able power on servers. Adding 1 and dividing by 2 ensures that the score will be positive between 0 and 1 (lines #20 and #22), while higher values are better.

Each run of the algorithm is for a single cluster with a few thousand homogeneous servers (heterogeneity is across clusters). Overall, the algorithm takes only 7 milliseconds to run, which is negligible as VM creation takes many seconds [1].

3.4 Per-VM power capping controller

To protect performance-critical VMs without losing power safety, we augment Azure’s out-of-band (i.e., independently of software on the server) full-server capping mechanism with an in-band (i.e., software-only) controller to cap only non-user-facing VMs when necessary. In our modified system, the chassis manager polls the PSUs every 200ms and alerts the in-band controller on each server when the chassis power draw is close to the chassis budget. Upon an alert, each controller uses per-core DVFS to power-cap the cores running non-user-facing VMs. To account for (1) high power draws between polls or (2) the inability of the controller to bring power below the budget, we keep the out-of-band mechanism that uses RAPL to throttle all cores equally as a backup.

To manage power per VM, we use the hypervisor’s core-grouping feature (e.g., cpupools in Xen, cpugroups in Hyper-V) to split the cores into high- and low-priority classes. We assign the user-facing VMs and the I/O VM (e.g., Domain0 in Xen, Root VM in Hyper-V) to run on cores in the high-priority class, and the non-user-facing VMs on the low-priority one.

Upon receiving an alert from the chassis manager, the per-VM power controller compares the server’s power draw to its budget. If the current draw is higher than the budget, the controller immediately lowers the frequency of the low-priority cores to the minimum p-state, i.e. half of the maximum frequency; the lowering of the frequency may entail a lower voltage as well. The goal is to quickly bring the server’s power draw below the limit and thereby avoid having to engage RAPL, which throttles all cores and impacts performance of all the VMs on the server. However, this large frequency reduction may overshoot the needed power reduction. To reduce the impact on the non-user-facing VMs, the controller then enters a feedback loop where each iteration involves (1) checking the server power meter and (2) increasing the frequency of N low-priority cores to the next higher p-state (100 MHz step), until the power is close to the budget. It selects the highest frequency that keeps the power below this threshold. $N = 4$ works well in our experiments. The feedback loop also adapts to changes in workload behavior on the VMs, which ends up impacting the server power draw.

It is possible that cutting the frequency of the low-priority cores in half is not enough to bring the power below the server’s budget. For example, a VM placement where there are not enough non-user-facing VMs in the workload mix, non-

user-facing VMs exhibiting lower utilization than predicted, or a controller bug can cause this problem. In this case, the out-of-band mechanism will kick in as backup. More aggressive mechanisms to reduce power, such as core sleep states (c-states) or shutting down the non-user-facing VMs, can also be leveraged before resorting to using RAPL. We will add such capability to our production system (Section 5).

The controller lifts the cap after some time (30 secs by default), allowing all cores to return to maximum performance.

3.5 Oversubscription strategy

We now describe our oversubscription strategy, which uses our per-VM capping system and placement policy, historical VM arrivals, and historical power draws, to increase server density. We considered using the placement policy along with Azure’s existing full-server capping system for oversubscription. However, without per-VM throttling, the only way to protect performance-critical VMs is to avoid capping events on servers/chassis running these VMs. This drawback severely limits the level of oversubscription.

Our strategy uses the algorithm below for computing an aggressive power budget for all the chassis of each hardware generation. Adapting it to find budgets for larger aggregations (e.g., rack, row) is straightforward. We refer to the uncapped, nominal core frequency as the “maximum” frequency.

To configure the algorithm, we need to select the maximum acceptable rate of capping events (e.g., #events per week) for user-facing ($emax_{UF}$) and non-user-facing ($emax_{NUF}$) VMs, and the minimum acceptable core frequency (e.g., half the maximum frequency) for user-facing ($fmin_{UF}$) and non-user-facing ($fmin_{NUF}$) VMs. If we want no performance impact for user-facing VMs, we set $emax_{UF} = 0$ and $fmin_{UF} =$ maximum frequency. As we describe next, our 5-step algorithm finds the lowest chassis power budget that satisfies $emax_{UF}$, $emax_{NUF}$, $fmin_{UF}$, and $fmin_{NUF}$.

Estimate future behaviors based on history:

1. Estimate the historical average ratio of user-facing virtual cores in the allocated cores (β). Estimate the historical average P95 utilization of virtual cores in user-facing ($util_{UF}$) and non-user-facing ($util_{NUF}$) VMs.

Profile the hardware:

2. Estimate how much server power can be reduced by lowering core frequency at $util_{UF}$ and $util_{NUF}$, given $fmin_{UF}$ and $fmin_{NUF}$, respectively. This step produces two curves for power draw (one curve for each average utilization), as a function of frequency.

Compute power budgets based on historical draws:

3. Sort the historical chassis-level power draws (one reading per chassis per unit of time) in descending order.

4. Start from the highest power draw as the first candidate budget and progressively consider lower draws until we find P_{min} . For each candidate power budget, we check that the rate of capping events would not exceed $fmax_{UF}$ or $fmax_{NUF}$ (considering the higher draws already checked), and the at-

tainable power reduction from capping is sufficient (given β and the curves from step 2).

5. To compute the final budget, add a buffer (e.g., 10%) to the budget from step 4 to account for future variability of β or substantial increases in chassis utilization.

We can use the difference between the overall budget computed in step 5 and the provisioned power to add more servers to the datacenter. Because we protect user-facing VMs and use our VM scheduling policy, this difference is substantially larger than in prior approaches, as we show in Section 4.6.

Example. Suppose (1) we are willing to accept rates of 0.1% and 1% capping events for user-facing and non-user-facing VMs, respectively; (2) upon a capping event, we are willing to lower the core frequencies to 75% and 50% of the maximum, respectively; (3) we have 10000 historical chassis power draws (collected from every chassis); and (4) the highest draws have been 2900W, 2850W, and 2850W. We first consider 2900W. If we were to set the chassis budget to just below that value to say 2890W, there would be 1 capping event out of 10000 observations, i.e. a rate of 0.01%, and we would have to shave 10W during the event. Given the acceptable capping rates and minimum frequencies, we can operate with the data from step 1 and the curves from step 2 to determine (a) whether we could reduce power by 10W, and (b) whether there would be an impact on user-facing VMs. If we can achieve the reduction, we count 1 event out of 10000 that would affect non-user-facing VMs. If the user-facing VMs would also have to be throttled, we would count 1 event out of 10000 that would affect those VMs. Since both rates are lower than 0.1% and 1%, we can now check a budget just below 2850W, say 2840W. We repeat the process for this budget, then the next lower budget and so on, until we violate the desired capping rates and minimum frequencies.

4 Evaluation

4.1 Methodology

Data analysis. We evaluate our criticality algorithm and ML models (Section 4.2) using standard metrics, such as precision and recall from predictions. We compute the metrics based on Azure’s *entire VM workload in April 2019*.

Real experiments. We run experiments on the same hardware that Azure uses in production. We use a chassis with 12 servers, each containing 40 cores split into two sockets. At their nominal frequency, each server draws between 112W (idle) and 310W (100% CPU utilization). At half this frequency, each server draws from 111W to 169W.

Our single-server experiments (Section 4.3) explore our per-VM capping controller and resulting VM workload performance for a combination of user-facing and non-user-facing VMs and various power budgets. For comparison, we use the existing full-server capping controller (RAPL) in Azure. Our chassis-level experiments (Section 4.4) explore our system on 12 servers, including PSU alerts. For comparison, we use Azure’s existing chassis-level mechanisms.

Parameter	Value
Cluster configuration	20 racks \times 3 chassis \times 12 blades
Blade configuration	2x20 cores
VM size dist. (cores)	1 (33%), 2 (27%), 4 (21%), 8 (10%), 16 (5%), 24 (3%), ≥ 32 (1%)
Deployment size dist. (#VMs)	1 (39%), 2 (14%), 3-5 (16%), 6-10 (9%), 11-15 (8%), 16-25 (5%), >25 (9%)
VM lifetime dist. (hours)	1 (52%), 2 (5%), 3-5 (10%), 6-10 (9%), 10-25(7%), 26-720 (8%), >720 (9%)
Workload type buckets	user-facing (UF), non-user-facing (NUF)
P95 utilization buckets	0-25%, 26-50%, 51-75%, 76-100%
Avg UF:NUF core ratio	4:6
Avg UF and NUF P95 util	65% (bucket #3), 44% (bucket #2)
# simulation days	30

Table 1: Simulation parameters.

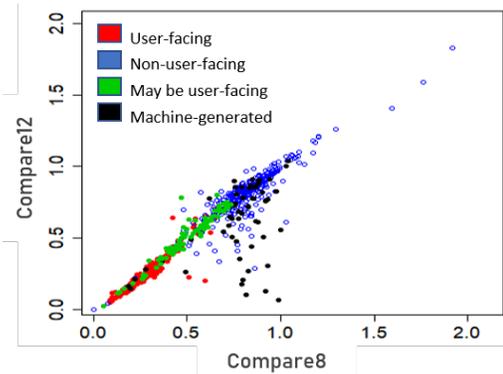


Figure 2: Algorithm compared to manual classification.

For both sets of experiments, we use instances of a latency-critical transaction processing application (similar to TPC-E) for the user-facing workload, and instances of a batch Hadoop computation (Terasort) for the non-user-facing workload. For the user-facing workload, we use real inputs from the team responsible for it, whereas we use synthetic input data for the non-user-facing computation.

Simulation. We evaluate our modified VM scheduler in simulation (Section 4.5), leveraging the same simulator that Azure uses to evaluate changes to the VM scheduler before putting them in production; our only extension is to simulate calls to Resource Central. An event generator drives the simulation with a sequence of VM arrivals. For each arrival, it invokes the production scheduling algorithm (or the scheduling algorithm with the addition of our policy) for server placement decisions. Running the actual scheduler code in the simulator ensures that simulations are faithful to reality.

We simulate a cluster of 60 chassis in 20 racks. The simulator produces VM arrivals based on distributions matching Azure’s load in April 2019. Table 1 lists the main statistics.

4.2 Criticality algorithm and ML models

Criticality algorithm. To evaluate our algorithm (Section 3.2), we first compare its classifications to our own manual labeling of 840 workloads. Figure 2 shows one dot for each workload with coordinates corresponding to its Compare8 and Compare12 values. The colors indicate whether we deem the workload clearly user-facing, possibly user-facing,

Technique	Recall target	Recall achieved	Precision achieved
Pattern-matching	99%	99%	76%
ACF	99%	99%	54%
FFT	99%	99%	48%
Pattern-matching	98%	98%	77%
ACF	98%	98%	56%
FFT	98%	98%	50%

Table 2: Pattern-matching vs ACF vs FFT.

clearly machine-generated, or clearly non-user-facing. The figure shows that Compare8 can separate the first two groups, which the algorithm should conservatively classify as user-facing, from the last two. A vertical bar at Compare8=0.72 gets all important workloads to the left of the bar, and the vast majority of unimportant ones to the right. Compare12 does not separate the classes well.

Thus, the algorithm accurately classifies workloads based on their Compare8 value. For a quantitative assessment, we compare it to two well-known approaches for finding periodicity in a time series, ACFs and FFTs, for the same set of workloads. For both approaches, we do the same pre-processing and disambiguate between user-facing and machine-generated workloads using the same methods as in our algorithm.

As we want to protect user-facing VMs, we must achieve high recall for this class as the recall indicates the probability of correctly identifying these VMs. Table 2 shows the precision and recall, for two high recall targets (0.99 and 0.98) for whether a workload is user-facing. Our algorithm achieves the target recall with much higher precision, i.e. it classifies many more non-user-facing VMs correctly. This reduces performance degradation during capping events, as more of those VMs can be throttled to lower power.

ML models. We now evaluate our models for *Azure’s entire VM workload*. Table 3 lists the percentage of predictions with confidence score higher than 60% (3rd column), and the per-bucket recalls and precisions (4th-7th columns) and the accuracy (rightmost column) for those high-confidence predictions. The VM scheduler disregards predictions with lower confidence and conservatively assumes the VM being deployed will be user-facing and will exhibit 100% 95th-percentile utilization. For comparison, we show results for the equivalent Gradient Boosting (GB) models.

The table shows that our criticality model achieves 99% recall for user-facing VMs (Bucket 2), which is critical for protecting these VMs. The most important features for our model are the percentage of user-facing VMs observed in the cloud subscription, the percentage of VMs that live longer than 7 days in the subscription, and the total number of VMs in the subscription. The GB model achieves similar results.

Our utilization model also does well with good recall and precision (83-93%) for the most popular buckets (1 and 4), and good accuracy (84%) for the 73% of high-confidence predictions. Here, the most important features are the average of the VMs’ 95-percentile CPU utilizations in the subscription, the average of the VMs’ average CPU utilizations in

Prediction	Model	% High Conf.	Bucket 1 R P	Bucket 2 R P	Bucket 3 R P	Bucket 4 R P	Accuracy
Criticality	GB	99%	67% 77%	99% 99%	NA	NA	98%
	RF	99%	69% 78%	99% 99%	NA	NA	98%
P95 util	GB	68%	95% 85%	47% 77%	51% 79%	94% 80%	82%
	RF	73%	93% 87%	61% 76%	65% 81%	92% 83%	84%

Table 3: Random Forest (RF) and Gradient Boosting (GB) models recall (R), precision (P), and accuracy for high-confidence predictions.

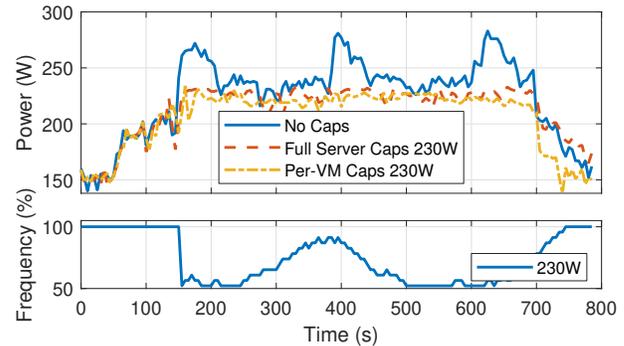


Figure 3: Server power dynamics.

the subscription, and the percentages of VMs in each CPU utilization bucket in the subscription. The GB model achieves similar accuracy, but with fewer high-confidence predictions and lower recall for the two middle (least popular) buckets.

4.3 Per-VM capping controller experiments

We run experiments on a server with our user-facing application running on a VM with 20 virtual cores and our non-user-facing application running simultaneously on another VM with 20 virtual cores. Each execution takes 10 minutes.

Figure 3 plots the dynamic power behaviors and core frequencies of full-server and per-VM capping with caps at 230W. In the bottom graph, we plot the lowest frequency of any non-user-facing core. The experiments have capping enabled throughout their executions. For comparison, we show the power profile of an experiment without any cap.

When unconstrained (no cap), the power significantly exceeds 250W. In contrast, full-server and per-VM capping keep the power draw below 230W. Because of the lower target of our controller (225W for the 230W cap), its draws are slightly below those of full-server capping most of the time. The frequency curve depicts the adjustments that our controller makes to the performance of the non-user-facing VM. The steep drop to the lowest frequency occurs when the controller abruptly lowers the frequency to the minimum value when the power first exceeds the target. After that, its feedback component smoothly increases and decreases the frequency.

Figure 4 shows the impact of capping in these experiments on the 95th-percentile latency of the user-facing application (10 leftmost bars) and the running time of the non-user-facing application (10 rightmost bars). Results are normalized to the unconstrained performance of each application. We also include bars for capping at 250W, 240W, 220W, and 210W.

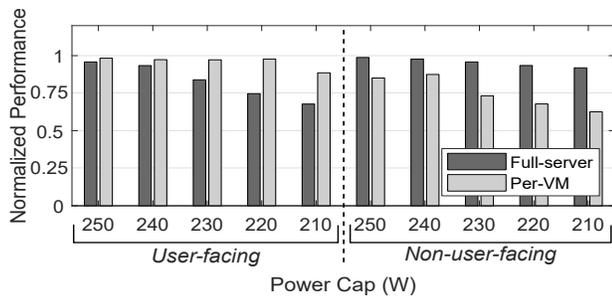


Figure 4: Performance impact of power capping.

The results show that full-server capping imposes a large tail latency degradation, especially for the lower caps. When the cap is 230W, the degradation is already 18%, which is often unacceptable for user-facing applications. For lower caps, full-server capping provides even worse tail latency (35% degradation for 210W). In contrast, our controller keeps tail latency very close to the unconstrained case, until the cap is so low (210W) that it becomes impossible to protect the user-facing application and RAPL needs to engage. This positive result comes at the cost of performance loss for the non-user-facing application. While full-server capping keeps running time fairly close to the unconstrained case, our controller degrades it by 28% for the 230W cap. This is the right tradeoff, as non-user-facing workloads have looser performance needs.

4.4 Chassis-level capping experiments

We now study the power draw at the chassis level, and the impact of different capping granularities (full-server vs per-VM) and VM placements. We experiment with a 12-server chassis running 36 copies of our user-facing application (each on a VM with 4 virtual cores), and 36 copies of our non-user-facing application (each running on a VM with 6 virtual cores). In terms of VM placement, we explore two extremes: (1) *balanced* placement, where we place the user-facing and non-user-facing VMs in round-robin fashion across the servers, i.e. 3 VMs of each type on each server; and (2) *imbalanced*, where we segregate user-facing and non-user-facing VMs on different sets of servers. Each experiment runs for 26 minutes.

Figure 5(left) plots the dynamic behavior of the chassis for an overall budget of 2450W for the two capping approaches. For comparison, we also plot the no-cap case. In these experiments, we use the balanced placement as an example.

As expected, both capping granularities are able to limit the power draw to the chassis budget, whereas the no-cap experiment substantially exceeds this value. We observe the same trends under the imbalanced placement approach. The placement does not matter in terms of the power profiles because the capping enforcement ensures no budget violations.

However, VM placement has a large impact on application performance. Figure 5(right) plots the impact of VM placement and capping granularity on the average 95th-percentile latency of the user-facing applications (4 leftmost bars) and on the average running time of the non-user-facing applications (4 rightmost bars). We normalize to the no-cap results.

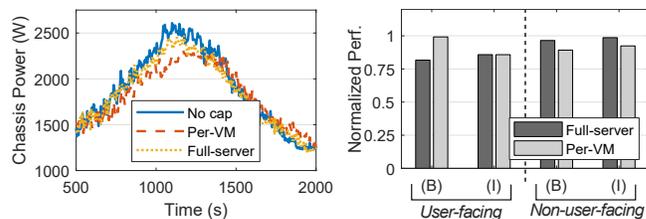


Figure 5: Chassis dynamics and performance vs placement.

Per-VM capping under balanced placement keeps the average tail latency the same as the no-cap experiment, despite the tight 2450W budget. In contrast, per-VM capping degrades performance as much as full-server capping when the placement is imbalanced. These results show that our controller protects user-facing VMs when the placement allows it.

Full-server capping provides slightly better performance than per-VM capping for the non-user-facing applications. More interestingly, the results for balanced placement are slightly worse than for imbalanced placement, regardless of the capping granularity. For per-VM capping, the reason is that servers with only non-user-facing VMs need to reduce the frequency of fewer cores. For full-server capping, the reason is that servers with only non-user-facing VMs tend to have higher utilization, so a smaller reduction in frequency is enough for a large power reduction. Comparing the two rightmost bars, we see that full-server capping hurts performance slightly less than per-VM capping in the imbalanced case, as RAPL lowers frequency more slowly than our controller.

4.5 Cluster VM scheduler simulation

In the previous section, we explored extreme and manually-produced VM placements in controlled experiments. However, in practice, placements are determined by the VM scheduler. Consequently, we implement our placement policy (Algorithm 1) as a preference rule in Azure’s VM scheduler, and simulate a cluster using 30 days of VM arrivals (Section 4.1). The simulator runs the same rules as in production, along with our added preference rule. It reports four main metrics:

- Deployment failure rate: percent of VM deployment requests rejected due to resource unavailability or fragmentation. This rate impacts users, so our policy should not increase it;
- Average empty server ratio: percent of servers without any VMs averaged over time. Empty servers can host the largest VM sizes, so our policy should not decrease this ratio;
- Standard deviation of the average chassis score, i.e. $1 - (\rho^{Peak} / \rho^{Max})$ (Algorithm 1, line 13), for each chassis. This metric reflects how balanced the chassis are with respect to their power loads. Lower values are better and mean better balance and fewer power capping events;
- Standard deviation of the average server score, i.e. $(1/2) \times (1 + (\gamma^{NUF} - \gamma^{UF}) / N^{cores})$ (Algorithm 1, line 20), for each server. This metric reflects how balanced the servers are in terms of UF and NUF core 95th-percentile utilizations. Lower values mean better balance and that we are more likely to only need to cap NUF VMs.

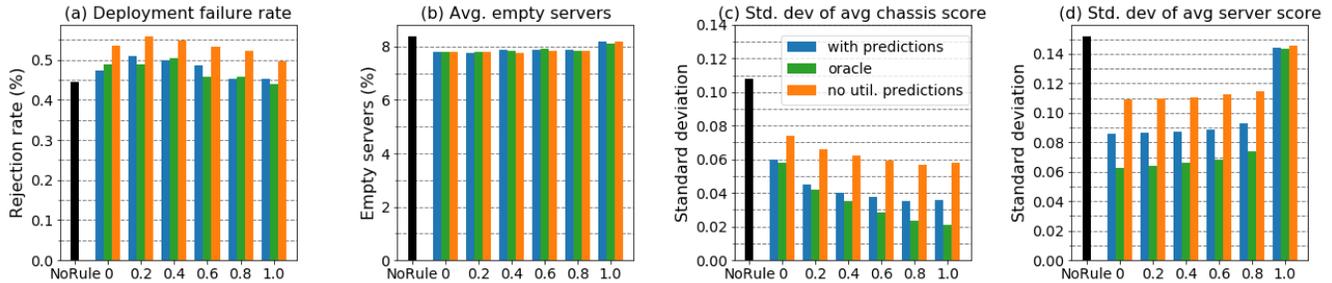


Figure 6: Key scheduler metrics, as a function of α .

Results. Figure 6 shows the results for these metrics, as a function of the α weight in our policy (Algorithm 1, line 6). $\alpha = 1$ means that the server score is irrelevant, whereas $\alpha = 0$ means that the chassis score is irrelevant. From left to right in each graph, the “NoRule” (black) bar represents the existing scheduler; the leftmost (blue) bar in each group represents our modified scheduler using our policy and ML predictions; the next (green) bar shows the modified scheduler using oracle predictions; and the rightmost (orange) the modified scheduler using criticality predictions, but no utilization predictions.

Comparing the black and blue bars illustrates the benefit of our policy and predictions. Figures 6(a) and (b) show that our modified scheduler impacts the failure rate slightly for low values of α (not at all for high values), while slightly decreasing the percentage of empty servers regardless of α . The reason is that our policy may use a few more servers in the interest of better balancing the load. In fact, Figures 6(c) and (d) confirm that the load is more balanced using our policy and predictions. These latter graphs also show that the value of α is important. $\alpha = 0$ produces much worse utilization balancing across chassis than other values. At the same time, $\alpha = 1$ produces as poor server utilization balancing as the existing scheduler, whereas other values produce much better server balancing. These observations confirm that it is key to balance both across chassis and servers, as in our policy. $\alpha = 0.8$ strikes a good compromise between the importance of these types of balancing.

Impact of prediction accuracy. Comparing the blue and green bars illustrates the impact of mispredictions and predictions with low confidence. Figures 6(c) and (d) show that oracle predictions produce only slightly better balancing than our real predictions for certain values of α .

Impact of criticality and utilization predictions. Comparing the orange and blue bars with the black bar illustrates the impact of having criticality only, and both criticality and utilization predictions, respectively. Clearly, it is critical to predict the workload type of each VM, as we want to protect the performance of user-facing VMs during capping events (black vs orange bars). The results demonstrate that having utilization predictions is also important (orange vs blue bars). The lack of such predictions degrades the balancing substantially for most values of α , and thus increases the capping rate and limits the power reduction during an event (thereby

decreasing the potential for oversubscription).

4.6 Oversubscription increases

We now estimate the amount of oversubscription and dollar savings that result from our lower chassis power budgets. To do so, we translate the amount of budget we can reduce in each 60-chassis cluster into the infrastructure cost we would avoid. Oversubscription allows servers to be added to an existing datacenter (assuming space, cooling, and networking are available, as it is often the case), avoiding the cost of building a corresponding fraction of a new datacenter.

We instantiate our 5-step oversubscription strategy (Section 3.5) with power telemetry from 1440 of Azure’s chassis over 3 months in 2018. We also use VM statistics from April 2019. Specifically, the 95th-percentile core utilizations for non-user-facing ($util_{NUF}$) and user-facing ($util_{UF}$) VMs were 44% and 65%, respectively, and the ratio of user-facing cores in the allocated cores (β) was 40%. Although user-facing VMs dominate in absolute count over the month, a majority of such VMs are short-lived – 52% of VMs last for less than 1 hour (Table 1). In contrast, all non-user-facing VMs last for at least 5 weekdays (Section 3.2). Consequently, a snapshot at various points in time in the month indicates an average value of 40% for β . We add a buffer of 10% to the chassis budget (step 5). For the results where providers treat external (i.e., third-party) VMs differently than internal (i.e., first-party) VMs, we adjust these parameters accordingly, while keeping the same amount of buffer.

Table 4 lists results for several types of provisioning:

1. “Traditional” provisioning (no oversubscription);
2. State-of-the-art full-server capping without VM insights.

In this approach, the power capping events need to be rare and the throttling has to be light to prevent performance loss to user-facing VMs. To model this approach with our provisioning strategy, we use $emax_{UF} + emax_{NUF} = 0.1\%$, $fmin_{UF} = fmin_{NUF} = 75\%$;

3. Predictions-based per-VM capping and scheduling, without impact on user-facing VMs. We use $emax_{UF} = 0$, $fmin_{UF} = 100\%$, $emax_{NUF} = 1\%$ and $fmin_{NUF} = 50\%$;

4. Predictions-based per-VM capping and scheduling, with minimal impact on user-facing VMs. To make this approach comparable to the others, we set the overall rate of capping events at 1%. Specifically, we use $emax_{UF} = 0.1\%$, $fmin_{UF} = 75\%$, $emax_{NUF} = 0.9\%$ and $fmin_{NUF} = 50\%$;

Approach	Chassis budget delta (%)	Savings (\$10/W)
Traditional	0	0
State of the art	6.2%	\$79.4M
Predictions for all VMs, no UF impact	11.0%	\$140.8M
Predictions for all VMs, minimal UF impact	12.1%	\$154.9M
Predictions for internal VMs, no UF impact	8.4%	\$107.5M
Predictions for internal VMs, minimal UF impact	10.3%	\$131.8M
Predictions for internal and non-premium external VMs, no UF impact	10.6%	\$135.7M
Predictions for internal and non-premium external VMs, minimal UF impact	12.1%	\$154.9M

Table 4: Comparison between provisioning approaches.

5. Predictions-based per-VM capping and scheduling for internal VMs only (all external VMs considered user-facing), without impact on user-facing VMs;

6. Predictions-based per-VM capping and scheduling for internal VMs only, with minimal impact on user-facing VMs;

7. Predictions-based per-VM capping and scheduling for internal and non-premium external VMs, without impact on user-facing VMs; and

8. Predictions-based per-VM capping and scheduling for internal and non-premium external VMs, with minimal impact on user-facing VMs.

The state-of-the-art approach achieves 6.2% oversubscription. This amount is comparable to that (8%) achieved by Facebook [43], which knows the (single) workload that runs on each server. Public cloud platforms do not have this luxury.

In contrast, our approach (#3 and #4) can *almost double* the oversubscription and savings. We achieve more than 12% oversubscription with minimal impact on user-facing VMs. Assuming a datacenter campus of 128MW and an infrastructure cost of \$10/W [4], 12.1% oversubscription translates into \$154.9M in savings; an increase in savings of \$75.5M over the state of the art. As providers can oversubscribe many campuses, the savings would be much higher in practice.

When providers prefer to treat external and internal VMs differently (approaches #5-#8), they can do so at the cost of a lower increase in oversubscription. For example, when treating all external VMs as user-facing and protecting the performance of user-facing VMs (#5), the increase in savings becomes \$28.1M. At the other extreme, where we treat the premium external VMs as user-facing and allow minimal impact on user-facing VMs (#8), the increase in savings returns to \$75.5M. The reason is that this provisioning approach has enough non-critical VMs, and oversubscription is limited only by the rate of capping events.

As our results show, the amount of oversubscription and savings depends upon the user-facing vs non-user-facing core ratio (β) and the amount of power that can be recovered from

each type through frequency reduction (f_{min}), while satisfying the constraints (e_{max}). Generally, a higher (lower) value of β results in less (more) oversubscription. This analysis for oversubscription has to be done on a per-cluster basis.

5 Lessons from production deployment

We have deployed our per-VM capping controller and ML models on thousands of servers in multiple datacenters. Next, we discuss some of the lessons from these deployments.

Hypervisor support for per-VM power capping. Our prototype controller (Section 3.4) leveraged the hypervisor’s core-grouping feature to manage the frequency of each VM’s physical cores. In production, Azure typically prefers not to restrict a VM to a subset of cores, so we could not rely on this feature. Instead, we had to extend the hypervisor to (1) add the capability to dynamically specify the frequency for a VM, and (2) carry the frequency to whichever cores it schedules the VM on during the context switch (changing the frequency takes tens of microseconds, whereas a scheduling quantum lasts 10 milliseconds). As most VMs are small (Table 1), there was no need to manage frequency on a per-virtual-core basis.

Refresh VM criticality prediction on servers. Misclassification of a VM’s criticality can result in unintended performance degradation (Section 4.2). To address this problem, we added the capability to periodically (e.g., daily) refresh a VM’s criticality tag on a server. We change the tag after Resource Central has observed the VM long enough to classify the criticality of the VM’s workload. As the prediction models already provide good placements (Section 4.5), we do not migrate VMs when their criticality changes.

Expanded workload criticality definition. Some first-party customers were concerned about the impact of per-VM capping on their non-user-facing VMs. To alleviate their concerns, we added a configurable *prioritized* throttling list to our system. Using the list, we first consider all low-priority and internal non-production VMs for throttling and throttle production non-user-facing VMs as a last resort, i.e. when throttling the other types is insufficient. Furthermore, our system has a “do-not-throttle” list of highly-sensitive internal workloads (e.g., gaming, repair) that are always considered critical. Finally, the criticality of VMs can be also be dynamically updated on servers based on changes to the static lists.

Metrics to measure capping impact. Since VMs are black boxes, we cannot use any workload-specific metric in production. Instead, our deployed system measures how long and how hard VMs are being capped. The data shows that our system is successful at protecting production user-facing VMs, while prioritizing the VMs that do get throttled.

Increasing rack density with per-VM capping. While deploying our system, we learned that Azure was installing fewer servers per rack when deploying a new generation of power hungrier servers. Having fewer servers per rack reduces the probability that the rack power draw will hit the provisioned limit and cause capping using RAPL. With our

per-VM capping system in place, Azure can increase the number of servers per rack. *This is another type of power oversubscription that per-VM capping enables.*

Shutting down VMs. Some first-party customers indicated that they would prefer their VMs to be shut down rather than throttled, as their services can handle losing VMs but an unpredictable impact due to throttling is not acceptable. Under extreme power draws, shutting down these customers' VMs can help protect production user-facing VMs and throttle fewer non-user-facing ones. We will soon add this capability to our system.

Server support for per-VM management. Our production experience has highlighted the drawbacks of managing VM power per-component (e.g., core, uncore, memory). We expect that cloud providers would prefer to raise the level of abstraction from individual components to entire VMs, even if VM power would have to be approximated. This would enable advances that have been too complex for production use, such as power-aware VM placement, enforcing per-VM power limits, and making throttling and shutting down decisions based on VM power. We are working with silicon vendors towards this end.

6 Related work

Our paper is *the first to use ML predictions for increasing power oversubscription in public cloud platforms*. Next, we discuss some of the most closely related works.

Leveraging predictions. Some works predict resource demand, resource utilization, or job/task length for provisioning or scheduling purposes, e.g. [7, 9, 13, 20, 23, 37]. In contrast, we introduce a new algorithm and ML models for predicting workload type and high-percentile utilization, seeking to protect critical workloads from capping and place VMs in a criticality- and capping-aware manner.

Server power capping. Most efforts have focused on selecting the DVFS setting required to meet a tight power budget as applications execute, e.g. [16, 19, 21, 22, 26, 29–31, 34, 36, 45]. Both modeling/optimization and feedback techniques have been used. The inputs to the selection have been either application-level metrics (e.g., request latency), low-level performance counters, or operator annotations (e.g., high priority application). Our capping controller uses per-core DVFS and feedback, so it adds to this body of work. However, it also uses predictions about the VMs' performance-criticality as its inputs. Our approach seems to be the best for a cloud platform, since criticality information and application-level metrics are typically not available, and collecting and tracking low-level counters at scale involves undesirable overhead.

Controlling CPU bandwidth (or utilization) to cap server power has also been studied [8, 27]. Reducing CPU bandwidth allows cores to go into sleep states. This approach is complementary and can be used in conjunction with per-core DVFS in our server power capping system.

Cluster-wide workload placement/scheduling. Many

works select workload placements to reduce performance interference or energy usage, e.g. [5, 6, 11, 35, 41, 44]. Unfortunately, they are often impractical for a cloud provider, relying on extensive profiling, application-level metrics, short-term load predictions, and/or aggressive resource reallocation (e.g., via live migration). Live migration is particularly problematic, as it retains contended resources, may produce traffic bursts, and may impact VM availability; it is better to place VMs where they can stay. Our scheduler uses predictions in VM placement. Unlike prior work, it reduces the number and impact of capping events, and increases power oversubscription.

Datacenter oversubscription. Researchers have proposed to use statistical oversubscription, where one profiles the aggregate power draw of multiple services and deploy them to prevent correlated peaks [12, 14, 18, 38, 42]. Our work extends these works by using predictions to place the workload, inform capping, and increase oversubscription. Our oversubscription strategy is also the first to carefully control the extent and impact of capping on important cloud workloads. Also, unlike [38], it does not affect workload availability.

Others have studied hierarchical capping in production datacenters [12, 18, 43]. Our paper focuses on chassis-level power budget enforcement to make our experimentation easier. However, our techniques extrapolate directly. For example, for row-level budget enforcement, we can place VMs across rows trying to balance rows and servers.

Finally, researchers have proposed using energy storage to shave power peaks in oversubscribed datacenters [15, 24]. When peaks last long, this may require large amounts of storage, which our work does not require. Nevertheless, the two approaches are orthogonal and can be combined.

7 Conclusions

We proposed prediction-based techniques for increasing power oversubscription in cloud platforms, while protecting important workloads. Our techniques can increase oversubscription by $2\times$. We discussed lessons from deploying our techniques in production. We conclude that recent advances in ML and prediction-serving systems can unleash further innovations in cloud resource provisioning and management.

Acknowledgements

We thank the reviewers and our shepherd, Avani Wildani, for helping us improve this paper. We also thank Tristan Brown, Jayden Chen, Cheng Chi, Eric Lee, Roberto de Oliveira, Sudharssun Subramanian, and Brijesh Warriar for their suggestions and help in deploying per-VM capping in production.

References

- [1] Samiha Islam Abrita, Moumita Sarker, Faheem Abrar, and Muhammad Abdullah Adnan. Benchmarking vm startup time in the cloud. In *Benchmarking, Measuring, and Optimizing*, 2019.

- [2] Amazon EC2. Amazon EC2 Auto Scaling, 2020. <https://aws.amazon.com/ec2/autoscaling/>.
- [3] Amazon Web Services. AWS Firecracker, 2018. <https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/>.
- [4] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 2018.
- [5] Anton Beloglazov and Rajkumar Buyya. Energy Efficient Resource Management in Virtualized Cloud Data Centers. In *Proceedings of the International Conference on Cluster, Cloud and Grid Computing*, 2010.
- [6] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic Placement of Virtual Machines for Managing SLA Violations. In *Proceedings of the International Symposium on Integrated Network Management*, 2007.
- [7] Rodrigo N Calheiros, Enayat Masoumi, Rajiv Ranjan, and Rajkumar Buyya. Workload Prediction Using ARIMA Model and its Impact on Cloud Applications' QoS. *IEEE Transactions on Cloud Computing*, 2015.
- [8] Chih-Hsun Chou, Daniel Wong, and Laxmi N. Bhuyan. Dynsleep: Fine-grained power management for a latency-critical data center application. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 2016.
- [9] 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 Symposium on Operating Systems Principles*, 2017.
- [10] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. RAPL: Memory Power Estimation And Capping. In *Proceedings of the International Symposium on Low-Power Electronics and Design*, 2010.
- [11] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.
- [12] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the International Symposium on Computer Architecture*, 2007.
- [13] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive Elastic Resource Scaling for Cloud Systems. In *Proceedings of the International Conference on Network and Service Management*, 2010.
- [14] Sriram Govindan, Jeonghwan Choi, Bhuvan Urgaonkar, Anand Sivasubramaniam, and Andrea Baldini. Statistical profiling-based techniques for effective power provisioning in data centers. In *Proceedings of the European conference on Computer systems*, 2009.
- [15] Sriram Govindan, Di Wang, Anand Sivasubramaniam, and Bhuvan Urgaonkar. Leveraging stored energy for handling power emergencies in aggressively provisioned datacenters. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.
- [16] Akhil Guliani and Michael M. Swift. Per-Application Power Delivery. In *Proceedings of the European Conference on Computer Systems*, 2019.
- [17] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM allocation service at scale. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2020.
- [18] Chang-Hong Hsu, Qingyuan Deng, Jason Mars, and Lingjia Tang. Smoothoperator: Reducing power fragmentation and improving power utilization in large-scale datacenters. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.
- [19] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the International Symposium on Microarchitecture*, 2006.
- [20] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical Prediction Models for Adaptive Resource Provisioning in the Cloud. *Future Generation Computer Systems*, 2012.
- [21] Kostis Kaffes, Dragos Sbirlea, Yiyan Lin, David Lo, and Christos Kozyrakis. Leveraging application classes to save power in highly-utilized data centers. In *Proceedings of the Symposium on Cloud Computing*, 2020.
- [22] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *Proceedings of the International Symposium on Microarchitecture*, 2015.

- [23] Arijit Khan, Xifeng Yan, Shu Tao, and Nikos Anerousis. Workload Characterization and Prediction in the Cloud: A Multiple Time Series Approach. In *Proceedings of the International Conference on Network and Service Management*, 2012.
- [24] Vasileios Kontorinis, Liuyi Eric Zhang, Baris Aksanli, Jack Sampson, Houman Homayoun, Eddie Pettis, Dean M Tullsen, and Tajana Simunic Rosing. Managing distributed ups energy for effective power capping in data centers. In *Proceedings of the International Symposium on Computer Architecture*, 2012.
- [25] K. Lange. Identifying shades of green: The specpower benchmarks. *Computer*, 2009.
- [26] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. Server-level power control. In *Proceedings of the International Conference on Autonomic Computing*, 2007.
- [27] Shaohong Li, Xi Wang, Xiao Zhang, Vasileios Kontorinis, Sreekumar Kodakara, David Lo, and Parthasarathy Ranganathan. Thunderbolt: Throughput-optimized, quality-of-service-aware power capping at scale. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2020.
- [28] Yang Li, Charles R Lefurgy, Karthick Rajamani, Malcolm S Allen-Ware, Guillermo J Silva, Daniel D Heimsoth, Saugata Ghose, and Onur Mutlu. A Scalable Priority-Aware Approach to Managing Data Center Server Power. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2019.
- [29] Yanpei Liu, Guilherme Cox, Qingyuan Deng, Stark C Draper, and Ricardo Bianchini. Fastcap: An efficient and fair algorithm for power capping in many-core systems. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2016.
- [30] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the International Symposium on Computer Architecture*, 2015.
- [31] Kai Ma, Xue Li, Ming Chen, and Xiaorui Wang. Scalable power control for many-core architectures running multi-threaded applications. In *Proceedings of the International Symposium on Computer Architecture*, 2011.
- [32] Microsoft Azure. Azure containers, 2020. <https://azure.microsoft.com/en-us/overview/what-is-a-container/>.
- [33] Microsoft Azure. Overview of autoscale in Microsoft Azure, 2020. <https://docs.microsoft.com/en-us/azure/azure-monitor/platform/autoscale-overview>.
- [34] Asit K Mishra, Shekhar Srikantaiah, Mahmut Kandemir, and Chita R Das. Cpm in cmips: Coordinated power management in chip-multiprocessors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.
- [35] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of the Annual Technical Conference*, 2013.
- [36] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. No "power" struggles: Coordinated multi-level power management for the data center. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.
- [37] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *Proceedings of the International Conference on Cloud Computing*, 2011.
- [38] Varun Sakalkar, Vasileios Kontorinis, David Landhuis, Shaohong Li, Darren De Ronde, Thomas Blooming, Anand Ramesh, James Kennedy, Christopher Malone, Jimmy Clidas, and Parthasarathy Ranganathan. Data center power oversubscription with a medium voltage power plane and priority-aware capping. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.
- [39] Ryuichi Sakamoto, Thang Cao, Masaaki Kondo, Koji Inoue, Masatsugu Ueda, Tapasya Patki, Daniel Ellsworth, Barry Rountree, and Martin Schulz. Production Hardware Overprovisioning: Real-world Performance Optimization Using an Extensible Power-Aware Resource Management Framework. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2017.
- [40] Schneider Electric. Circuit Breaker Characteristic Trip Curves and Coordination, 2001. <https://www.se.com/us/en/download/document/0600DB0105/>.
- [41] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pMapper: Power and Migration Cost Aware Application Placement in Virtualized Systems. In *Proceedings of the International Conference on Distributed Systems Platforms and Open Distributed Processing*, 2008.

- [42] Guosai Wang, Shuhao Wang, Bing Luo, Weisong Shi, Yinghang Zhu, Wenjun Yang, Dianming Hu, Longbo Huang, Xin Jin, and Wei Xu. Increasing large-scale data center capacity by statistical power control. In *Proceedings of the European Conference on Computer Systems*, 2016.
- [43] Qiang Wu, Qingyuan Deng, Lakshmi Ganesh, Chang-Hong Hsu, Yun Jin, Sanjeev Kumar, Bin Li, Justin Meza, and Yee Jiun Song. Dynamo: Facebook’s data center-wide power management system. In *Proceedings of the International Symposium on Computer Architecture*, 2016.
- [44] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the International Symposium on Computer Architecture*, 2013.
- [45] Huazhe Zhang and Henry Hoffmann. Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

Proactive Energy-Aware Adaptive Video Streaming on Mobile Devices

Jiayi Meng
Purdue University

Qiang Xu
Purdue University

Y. Charlie Hu
Purdue University

Abstract

Energy-aware app adaptation enables mobile apps to dynamically adjust data fidelity such as streaming video quality to meet a user-specified goal for battery duration. Traditional energy-aware app adaptation is reactive in nature where the operating system monitors the app energy drain and signals the app to adapt upon detecting energy drain deviation from the pre-specified energy budget which can cause high oscillation and poor quality-of-experience (QoE).

In this paper, we observe that modern power-hungry apps such as video streaming and offloading-based apps already come with sophisticated app adaptation to deal with resource changes such as network dynamics and propose proactive energy-aware adaptation where the user-specified energy budget is integrated with the app adaptation logic. The potential benefit of such an approach is that app energy drain adaptation is no longer an “after-effect”, and hence the approach is likely to reduce the oscillation in app adaptation and improve the app QoE.

In this paper, we study the design, implementation and performance tradeoffs of reactive and proactive energy-aware app adaptation in the context of one of the most power-hungry classes of mobile apps, ABR-based video streaming. Our study shows that proactive energy-aware ABR video streaming is easy to implement by leveraging the built-in adaptation of modern apps and can improve the QoE of reactive approach by 44.8% and 19.2% in streaming 360° videos to Pixel 2 and Moto Z3 phones under low power budget.

1 Introduction

For enriched user experience, modern mobile apps utilize a large number of power-hungry hardware components such as the CPU, GPU, WiFi and 4G, and hardware decoder and as a result draw significant amount of power. As a result, the user experience of such feature-rich apps is often limited by the shortened battery life from increased power draw [7, 8].

Energy-aware app adaptation [31, 88] exploits a key observation that many apps can reduce their power draw by reducing their data fidelity such as the size and quality of a video in video streaming apps or the filtering level of a map in navigation apps, and enables apps to dynamically adjust their data fidelity to meet a user-specified goal for battery duration, *e.g.*, a four-hour plane ride.

There are two components in building an energy-aware app adaptation system: energy accounting and control. First, an energy accounting subsystem is needed to monitor the

energy drain during the app execution and detect any significant deviation, *e.g.*, exceeding a threshold, from the user-specified energy budget. Second, the app needs to implement some adaptation logic to stay within the energy drain budget or correct energy drain deviation.

Traditional energy-aware app adaptation schemes such as PowerScope [31], ECOSystem [88], Cinder [68], and Nemesis [62] have mainly focused on system-level energy accounting and control, treating the app as a “black-box”. In particular, in such systems, the operating system (OS) monitors the app energy drain during app execution, and upon detecting energy drain deviation from the pre-specified budget either throttles the execution of the app process or threads or issues an upcall to the app to trigger app reactive adaptation. The benefit of such a black-box approach is simplified app implementation, as real-time energy accounting is provided as an OS service. The downside of such an approach is that the disintegrated and reactive nature prevents jointly optimizing the app QoE and meeting the energy drain budget. In particular, correcting energy drain deviation as an after-effect can lead to app fidelity oscillation and negatively affect the QoE.

In this paper, we make a key observation that compared to the mobile apps studied two decades ago [31], mobile apps today not only are more power hungry, but also often come with sophisticated adaptation built in to optimize the user-perceived QoE in reaction to network dynamics or other system constraints. For example, fidelity adaptation such as adaptive bitrate (ABR) is now widely adopted in video streaming systems [15, 38, 45–47, 54, 59, 61, 65, 69, 72–74, 76, 83, 85], and adaptive offloading of computation to edge servers has been proposed for deep learning enhanced tasks such as video analytics [21, 37, 51, 57, 66].

We argue that the built-in QoE optimization frameworks in such modern mobile apps naturally lend themselves to *integrated, proactive* energy-aware app adaptation where the energy drain budget is seamlessly integrated into the pre-existing QoE adaptation as a constraint. The key benefit of such an integrated, proactive approach is that app energy drain adaptation is no longer an after-effect and hence likely to reduce the oscillation in app adaptation and improve the app QoE. Compared to the reactive approach, the integrated approach faces two design challenges: (1) the app needs to predict the power consumption for each adaptation candidate beforehand, and (2) the app needs to incorporate the energy budget into its QoE optimization logic.

In this paper, we study the design, implementation, and performance tradeoffs of reactive and proactive energy-aware

app adaptation in the context of one of the most power-hungry classes of mobile apps, ABR-based video streaming [41,79,87,89]. We focus on a state-of-the-art ABR scheme, Robust MPC [85], that employs receding horizon control [19] to maximize the QoE for the next few video chunks based on the predicted network throughput in the moving horizon of video chunk downloading intervals.

We design ENERGY-AWARE ABR, an energy-aware ABR video streaming system to demonstrate how to address the two challenges in designing proactive energy-aware app adaptation. First, proactive energy-aware app adaptation requires a power predictor that can *predict* the average power draw of the streaming app in the next time interval in fetching a chunk of any candidate chunk quality. Traditional mobile device power models [22,23,26,70,87,91] cannot be used as they take component utilization logged as input which are not available *before* app execution. We propose a novel function-level power modeling methodology that accurately predicts the app energy drain in the next interval.

Second, integrating an energy budget into the app adaptation logic, in particular, the model predictive control (MPC)-based QoE optimizer for MPC-based ABR, faces a unique challenge. Unlike other constraints such as network throughput in the QoE optimization problem, *app energy drain is cumulative and hence elastic over time*. At any time interval during a streaming session, the app may have accumulated energy drain surplus or deficit from past time intervals, *e.g.*, from selecting low chunk formats limited by transient low network bandwidth. We explore several design options on how to integrate such energy surplus/deficit with the QoE optimization framework of MPC-based ABR controllers.

Since MPC-based controllers are notoriously hard to analyze [17,34,67], we evaluate the end-to-end performance of reactive and proactive energy-aware ABR video streaming designs using testbed experiments. We have implemented both designs on top of Puffer [11], an open-sourced video streaming platform, and evaluated different design options by streaming 360° videos from a media server to a mobile client, while varying the network condition using network traces from two large datasets, YTrace and FCC [4].

Our evaluation results show that (1) Our function-wise power predictor achieves low mean per-interval (2-second) energy prediction error of 4.87% (Pixel 2) and 5.86% (Moto Z3). (2) Under dummy power budget, *i.e.*, using the average power draw of the energy-oblivious ABR as the power budget, both proactive and reactive ENERGY-AWARE ABR achieve only slightly lower QoE than that of the energy-oblivious ABR, with proactive ENERGY-AWARE ABR achieving slightly higher QoE than reactive ENERGY-AWARE ABR. (3) Under low power budget, proactive ENERGY-AWARE ABR improves the QoE by 44.8% (Pixel 2) and 19.2% (Moto Z3) over reactive ENERGY-AWARE ABR. (4) The majority of the improvement comes from significantly reduced video quality variation component of the QoE, of 85.2% (Pixel 2) and 87.4% (Moto Z3), showing that

proactive ENERGY-AWARE ABR can effectively mitigate the oscillation drawback of reactive energy-aware adaptation.

In summary, this paper makes the following contributions:

- We show prior reactive energy-aware app adaptation can lead to app fidelity oscillation which can negatively affect user-perceived QoE.
- We propose to our knowledge the first proactive energy-aware app adaptation and show that it can be easily implemented by integrating user-specified energy budget with the built-in app adaptation logic of modern apps such as MPC-based ABR systems for video streaming.
- We present a novel function-wise power predictor that can be used for what-if power draw analysis needed in proactive energy-aware app adaptation, *e.g.*, ENERGY-AWARE ABR.
- We experimentally compare the end-to-end performance of reactive and proactive energy-aware adaptive streaming of 360° videos on Pixel 2 and Moto Z3 phones.
- Our results show proactive energy-aware video streaming improves the QoE by 44.8% (Pixel 2) and 19.2% (Moto Z3) over the reactive approach under low power budget.

To further the research on energy-aware app adaptation, we have open-sourced ENERGY-AWARE ABR implementation.¹

2 Motivation

To motivate how energy-aware adaptation can help to reduce app power draw and elongate battery duration, we performed a measurement study of one of the most power-hungry classes of apps, video streaming, with example apps such as Youtube and Netflix consistently ranked among the top 10 battery draining apps [8,12,13].

Experimental Setup. We streamed 6 popular panoramic videos in full screen mode in the Youtube app on a Pixel 2 phone, which was connected to a Monsoon power monitor to measure the total phone power draw during video streaming. The videos were categorized into 3 groups, slow, medium and high, based on the moving speed of the camera. Each video was encoded into different resolutions and frame rates, including (4K, 60FPS), (1440p, 60FPS), (1080p, 60FPS), (720p, 60FPS), (480p, 30FPS) and (360p, 30FPS). Table 1 shows the bitrates of the videos at different quality settings.

We streamed each video in Youtube at different qualities with and without screen on, respectively. When the screen was on, the brightness level was set at 60%. To prevent the app from directly retrieving video frames from the cache rather than through the Internet, we cleared the cache of Youtube app on the phone before each experiment.

To understand the power breakdown of 360° video streaming, we built component-wise power models [22,26,44,75,

¹<https://github.com/meng72/Proactive-Energy-Aware-Adaptive-Video-Streaming>

Table 1: Average bitrate (Mbps) of 6 panoramic videos at different resolutions and frame rates from Youtube.

Videos	4K/60	1440p/60	1080p/60	720p/60	480p/30	360p/30
Slow						
Planets [1]	18.10	6.20	2.06	1.01	0.35	0.19
Solar [6]	22.10	5.38	1.70	0.76	0.22	0.10
Medium						
Orleans [5]	25.70	12.00	3.72	2.13	0.61	0.33
Chicago [9]	25.60	10.60	3.41	1.93	0.58	0.31
High						
Monster [2]	26.20	12.60	4.20	2.49	0.71	0.39
Optical [10]	24.40	12.60	4.16	2.47	0.71	0.39

81, 87] for major hardware components in the Pixel 2 phone, profiled hardware component usage while streaming 360° videos at (4K, 60FPS) in Youtube on Pixel 2, and finally estimated the corresponding power draw for each hardware component using the power models.

Findings. Figure 1 shows the average power consumption of streaming the 6 panoramic videos at different quality settings over the Internet in Youtube. We derived the screen power by subtracting the total power of video streaming when the screen was off from that when the screen was on. We see that (1) 360° video streaming is power-intensive on modern mobile devices. Streaming videos at (4K, 60FPS) consumes the highest total power of 579.73mA.² At this rate, a fully charged Pixel 2 phone (with a 2700 mAh battery) will drop to 15% in 3 hours and 40 minutes when the Battery Saver mode will turn on. (2) Thanks to the OLED technology, the power consumption of the display is relatively small, only 41.66–50.36mA, or 7.0%–12.0% of the total power across different settings. (3) Reducing the data fidelity, *i.e.*, the video resolution and frame rate, can significantly lower the total app power draw. For example, the total power draw decreases by 33.1% from 579.73mA to 387.79mA when the video quality changes from (4K, 60FPS) to (360p, 30FPS).

Figure 2 shows the power breakdown of streaming 360° videos at (4K, 60FPS). We see that (1) the CPU, GPU and screen consume relatively low power, of 82.11mA (14.0%), 74.49mA (12.7%) and 46.92mA (8.0%), respectively; (2) in contrast, the NIC (network interface card) and hardware decoder, two primary hardware components involved in video streaming, dominate the total power draw of streaming, *i.e.*, 220.53mA (37.6%) and 162.47mA (27.7%), respectively. These results suggest that adapting the data fidelity of video chunks to control the power draw of networking and decoding will be effective in controlling the total power draw of the app.

3 Prior Work on Energy-aware App Adaptation

The large body of research on managing the energy drain of applications on mobile systems has mainly focused on

² In this paper, for power measurement we directly report the current drawn in milli-Amperes (mA); the actual power consumed would be the current drawn multiplied by 3.7V, the voltage supply of the battery. The smartphone batteries are rated using these metrics and hence are easy to cross reference.

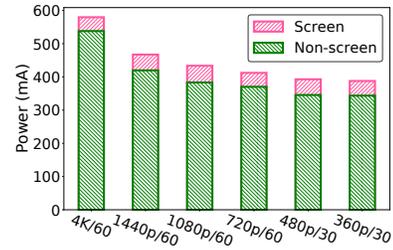


Figure 1: Average power consumption of streaming 360° videos with different qualities in Youtube on Pixel 2.

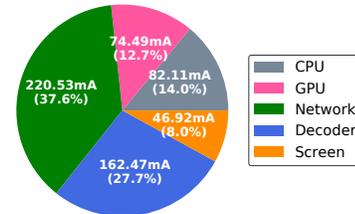


Figure 2: Power breakdown of streaming 360° videos at 4K/60FPS on Pixel 2.

system-level energy accounting and control, treating the app as a “black-box”.

Since managing the energy drain of apps requires accurately monitoring the energy drain during application execution, a large body of work proposed solutions to the resource container-level, process-level, or thread-level energy accounting problem in the OS, such as PowerScope [32], ECOSystem [88], Cinder [68], and Nemesis [62]. In a nutshell, energy accounting in these systems is achieved via either aligning external power measurement with interrupt-triggered program sampling as in early systems such as PowerScope [32] and Quanto [33], or using a pre-trained power model that captures the correlation between utilization of each hardware component in each of its power states and the resulting power draw and feeding the power model with hardware component usage logged during app execution to estimate the app energy drain, *e.g.*, in ECOSystem [88] and Cinder [68].

Upon detecting that an app’s energy drain has exceeded a predetermined budget, the system needs a way to throttle the app’s energy drain. In ECOSystem [88], Currentcy [88], and Cinder [68], the kernel would enforce energy budget by halting or throttling app threads, processes, or resource containers from execution. The Odyssey extension [31] and Nemesis [62] do not throttle applications, but issue upcalls or provide feedbacks to the applications to trigger fidelity adaptation to adjust their energy drain rate.

4 Reactive vs. Proactive Energy-aware App Adaptation

We discuss the drawbacks of prior reactive energy-aware app adaptation and propose proactive energy-aware app

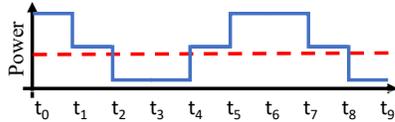


Figure 3: Reactive approach causes power oscillation.

adaptation.

Reactive energy-aware app adaptation. The prior system-level app energy control solutions are *reactive* and *disintegrated*. They are reactive because they treat applications as black-boxes and passively monitor their energy drain, and inform apps of the need to perform adaptation reactively upon detecting any deviation of the app energy drain from the pre-specified budget. They are disintegrated because the two tasks are performed in isolation: the OS monitors the app energy drain while the app performs adaptation.

The benefit of such a reactive approach is simplified app implementation, as real-time energy accounting can be provided as an OS service and the apps can focus on reactive adaptation, although fine-grained model-based energy monitoring relies on collecting fine-grained hardware component usage which can incur high runtime overhead.

The downside of a reactive approach is that the disintegrated and reactive nature deprives the opportunity of jointly optimizing the app QoE and meeting the energy drain budget at each time interval. In particular, the app not only performs adaptation *after* deviating from the energy drain target, but also typically does not have specific guidance on *how much* app fidelity to adapt in the next time interval, which can result in power draw and app fidelity oscillation.

Figure 3 shows an example of how the power draw of a disintegrated, reactive app adaptation scheme can cause oscillation in app power draw. Assume an app has three fidelities, high, medium, and low, drawing correspondingly high, medium, and low power. An energy-aware OS like Odyssey [31] tries to steer the app towards a target power budget specified in the dashed line to ensure a target battery duration. The app starts running in high fidelity, drawing high power. At t_1 , the OS performs an upcall informing the app of an energy drain deficit, and the app lowers the fidelity to medium. At t_2 , the OS informs the app of energy drain deficit again as the average power is still above the budget, and the app lowers the fidelity to low. The trend reverses in the next three intervals, and the oscillation would continue as a result of the reactive correction without concrete guidance on app adaptation due to the disintegrated approach.

Proactive energy-aware app adaptation. Our proposal of proactive energy-aware app adaptation is motivated by the above drawbacks of the reactive approach and an observation about modern mobile apps. Compared to the mobile apps studied two decades ago (e.g., [31]), mobile apps today are more power hungry, but also often come with sophisticated proactive adaptation built in to optimize the user-perceived

QoE under network dynamics or other system constraints. For example, proactive fidelity adaptation such as adaptive bitrate (ABR) (e.g., DASH) is now a standard feature in regular video streaming systems [15, 45, 46, 54, 59, 61, 72, 73, 76, 83, 85] as well as 360^o video streaming systems [38, 47, 65, 69, 74]. Similarly, adaptive offloading of computation to an edge server according to the network dynamics has been proposed in many systems [21, 37, 51, 57, 66]. More recently, adaptive offloading of machine learning inference has been proposed to adaptively offload a subset of DNN layers from the mobile device to the edge server [30, 49, 52, 92].

Motivated by the above two observations, we argue that the built-in QoE optimization frameworks in many modern mobile apps lend themselves to integrated, proactive energy-aware app adaptation that potentially overcomes the oscillation drawback of reactive approaches. In such an approach, at every step of the QoE optimization for selecting the quality of the chunk to be fetched in the next time interval, the power budget is explicitly taken into consideration so that the QoE optimization will directly output the optimal chunk format that maximizes the QoE for the next chunk and satisfies the energy budget in the next interval.

The key benefit of such an integrated, proactive approach is that app energy drain adaptation is no longer an after-effect correction and hence likely to reduce the oscillation in app adaptation. The challenge of such an approach is that each app needs to (1) predict the power consumption for each adaptation candidate beforehand, and (2) incorporate the energy budget in its QoE optimization logic.

In this paper, we investigate the design and performance tradeoffs of reactive and proactive energy-aware app adaptation in the context of one of the most power-hungry classes of mobile apps – ABR-based video streaming.

5 Energy-aware ABR

We briefly review state-of-the-art ABR algorithms and state the energy-aware QoE maximization problem.

Background on ABR. Adaptive bitrate (ABR) algorithms embody a primary technique of streaming videos over the Internet [71], where each video is encoded into multiple "tracks" with different quality bitrates and each track is segmented into "chunks" (e.g., 2-second each). These algorithms aim at optimizing the video QoE by dynamically selecting which chunk to fetch based on network conditions. Earlier ABR schemes were either "buffer-based" [45, 73], or "rate-based" which pivot on estimating available network throughput and finding a matching video bitrate [46, 76]. MPC [85] unifies the QoE objective of chunk k as a weighted sum of three key elements, (1) video quality, (2) video quality variation, and (3) stall time:

$$QoE_k = Q_k - \lambda|Q_k - Q_{k-1}| - \mu T_k \quad (1)$$

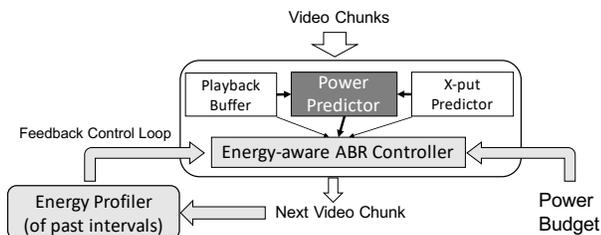


Figure 4: Architecture of energy-aware ABR. Lightly shaded components are new to both reactive and proactive designs, and the dark shaded one is new to the proactive design.

where Q_k represents the quality of video chunk k , T_k represents the stall time experienced by fetching chunk k and λ , and μ are weighting parameters for quality variation and rebuffering, respectively. Some papers (e.g., [85]) quantify Q_k as the bitrate of chunk k , while others (e.g., [83]) use perceptual quality metrics, e.g., SSIM [77]. The ABR problem is then formulated as maximizing the QoE of all the chunks of a video downloaded in a streaming session:

$$\text{maximize } \sum_k QoE_i, \text{ subject to buffer and network dynamics} \quad (2)$$

Since future throughput is unknown, practical algorithms like Robust MPC [85] employ receding horizon control [19] to maximize the QoE for the next few chunks (e.g., 5 future chunks). Such algorithms take estimated network throughput for the next few intervals and client playback buffer occupancy as input and output the quality format for the next video chunk to be downloaded from the server while maximizing the QoE of next few chunks.

Energy-aware QoE maximization problem. We consider the canonical energy-aware app adaptation scenario in previous work [31, 68, 88], where the phone user specifies the total energy budget E_b (e.g., 50% of batter level drop) over a fixed amount of time T_d (e.g., duration of a train ride) and hence an average power target $P_b = E_b/T_d$. The energy-aware ABR problem is then formulated as maximizing the QoE of all the chunks of a video downloaded in a streaming session:

$$\text{maximize } \sum_k QoE_i, \text{ subject to buffer and network dynamics} \\ \text{and total energy constraint} \quad (3)$$

Energy drain is elastic. We note that in the above problem statement, the energy constraint is different from other constraints like network throughput in that *app energy drain is cumulative and elastic over time*. Since the user-specified energy budget is the total energy drain over a streaming duration, the streaming app may accumulate some surplus or deficit based on the energy drain so far during the streaming session. For example, due to network bandwidth fluctuation, there can be intervals during which the network bandwidth is low and the ABR algorithm is forced to pick low video

Algorithm 1: Reactive Energy-Aware ABR – RA

Input : Power budget P_b ; Energy draw E_{actual} over streaming time so far T ; Selected format F_{k-1} for interval $k-1$; Average interval duration t

Output: Format F_k for next interval k

// if energy deficit, downgrade format

if $E_{actual} - P_b T > \gamma P_b t$ **then**

$F_k = \min(F_{k-1} - 1, \text{predicted format by ABR});$

else $F_k = \text{predicted format by ABR};$

quality formats which result in low power draw and hence low energy drain during those intervals. In such intervals, the app effectively accumulates an energy surplus, which can be spent in later intervals, i.e., when the network bandwidth is high and high quality video chunks can be downloaded and played, to improve the total QoE.

6 Reactive Energy-Aware ABR Design

A reactive energy-aware ABR can be easily implemented by adding an energy profiler and adding reactive adjustment to the QoE optimizer output of the ABR controller, as shown in Figure 4. The energy profiler monitors the app energy drain and is decoupled from the ABR controller, either implemented in the OS as in [36, 63, 86] or in an app-agnostic library linked with the app. It sends an upcall to the ABR player either reactively upon detecting significant deviation of the predetermined energy budget or periodically to inform the app of its energy drain so far which is used by the ABR controller to monitor the energy surplus or deficit and to decide when and how to adapt. We assume the later approach which gives the ABR controller more flexibility.

In particular, the ABR controller accumulates a running energy drain balance as the difference between the expected energy drain so far $P_b \cdot T$ and the actual energy drain E_{actual} so far which is monitored by the energy profiler.

The goal of the reactive adjustment is to adjust the chunk format selected by the ABR QoE optimizer to try to correct the energy drain deviation from that according to the pre-specified average power budget. To achieve this, the final chunk quality format is adjusted as no higher than the previous chunk format if there is an energy deficit greater than a threshold, as shown in Algorithm 1. We experimentally found a threshold of 10% of the energy budget to work well.

7 Proactive Energy-aware ABR Design

7.1 Integrated Energy-aware MPC Algorithm

Since an MPC-based ABR algorithm is proactive by nature, i.e., it calculates the video chunk quality to be fetched next based on the predicted network throughput in the next time interval, a proactive energy-aware ABR can be naturally realized by integrating the energy constraint with a practical

MPC algorithm such as Robust MPC [85]. In particular, the new energy-aware MPC algorithm optimizes the worst-case QoE assuming that the throughput in the future can take any value in range $[\hat{C}_t, \hat{C}'_t]$, by solving the following optimization problem at time t_k to derive the quality format for fetching the next chunk $F_k = f_{\text{empc}}(F_{k-1}, B_{k-1}, \hat{C}_t)$:

$$\max_{F_k, \dots, F_{k+N-1}} \min_{\hat{C}_t \in [\hat{C}_t, \hat{C}'_t]} \sum_k^{k+N-1} QoE_i$$

subject to buffer and throughput dynamics and

$$E_k + \dots + E_{k+N-1} < N \cdot P_b \cdot \delta t \quad (4)$$

where \hat{C}_t is the low bound in the predicted throughput range $[\hat{C}_t, \hat{C}'_t]$, B_{k-1} is the buffer occupancy after downloading chunk $k-1$ and δt is the interval duration, e.g., 2 seconds.

We note that since app energy drain is cumulative, converting the total energy constraint into a constant energy constraint per time interval can be conservative. As with MPC [85], we do not claim this energy-constrained MPC is necessarily the optimal control algorithm for the energy-constrained bitrate adaptation problem, but one that is practical and can leverage accurate network throughput prediction and power draw prediction in the near horizon.

7.2 Architecture Overview

Figure 4 (adding the power predictor) shows the architecture of our proposed integrated, proactive ENERGY-AWARE ABR. As with the original ABR, ENERGY-AWARE ABR derives the client-side playback buffer status and estimated network throughput from two generic modules of ABR, the playback buffer module and throughput predictor module, respectively. The energy profiler module estimates the energy drain in past intervals which is used to maintain the energy surplus/deficit.

To select the video format for the next video chunk, ENERGY-AWARE ABR uses a new power predictor to predict the average power draw of the next video chunk of each candidate format. Its controller then filters out the formats whose predicted energy drain exceeds the energy budget adjusted for energy surplus or deficit so far and chooses the one that maximizes the QoE stated in Equation 1.

The proactive ENERGY-AWARE ABR system design faces two challenges: (1) how to predict the power consumption in fetching future video chunks of different candidate formats in the power predictor? (2) how to incorporate the energy constraint into the MPC algorithm in the ENERGY-AWARE ABR controller to facilitate maximizing the QoE for the future chunks? We start with discussing our solution to (2).

7.3 Energy-aware QoE Maximization

The basic design for incorporating the expected energy drain for the N future intervals in Eqn. 4 is straight-forward. To exploit dynamic energy surplus/deficit, the ENERGY-AWARE

Algorithm 2: Design option 3 – LA(N)+LB

Input :Power budget P_b ; Energy draw E_{actual} over streaming time so far T ; Energy surplus/deficit $E_s = P_b T - E_{\text{actual}}$; Current buffer level B_{k-1} ; Predicted power array $P_k[f][N]$ for all the formats from next interval k to interval $k+N-1$; Interval duration δt

Output: Format F_k for next interval k

Call recursiveABR(0, 0, B_{k-1} , 0) to derive F_k ;

Function recursiveABR($n, F_{k+n-1}, B_{k+n-1}, E$):

```

if  $n == N$  then
  if  $E > N \cdot P_b \cdot \delta t + E_s$  then return  $-\infty$ ;
  else return Quality of  $F_{k+n-1}$ ;
end
 $\max\_QoE = -\infty$ ;
for  $i = 0$  to  $f$  do
   $q = \text{QoE between chunk } n \text{ and } n+1$ ;
   $B = \text{Buffer level after downloading chunk } n+1$ ;
   $E' = E + P_k[i][n] \cdot \delta t$ ;
   $Q = q + \text{recursiveABR}(n+1, i, B, E').\text{QoE}$ ;
  if  $Q > \max\_QoE$  then
     $F_{k+n} = i$ ;  $\max\_QoE = Q$ ;
  end
end
return  $\langle \max\_QoE, F_{k+n} \rangle$ 

```

ABR controller accumulates the energy surplus/deficit as in the reactive approach, which is then exploited by the QoE maximization module to maximize the QoE of future video chunks. We explore three design options for incorporating energy surplus/deficit into the QoE maximization.

(1) **Look ahead 1 (LA(1))**. The strawman design is to ignore energy surplus/deficit, and choose among all the candidate chunk formats with which the predicted app power draw in the next interval will not exceed the average power budget P_b , the one that maximizes the total QoE for the horizon (e.g., 5 intervals). Such a design can be conservative in terms of QoE from not exploiting potential energy drain surplus accumulated in the past intervals when the network bandwidth fluctuates up and down.

(2) **Look ahead 1 and look back (LA(1)+LB)**. Design option 2 extends LA(1) by exploiting the amount of energy surplus/deficit during past intervals. In selecting the video format for the new video chunk to fetch, the controller increases the energy budget for the next interval to $P_b \cdot \delta t + E_s$ (energy surplus/deficit). However, such a design may sacrifice video smoothness, as the energy surplus from past intervals may be large and allow some high quality chunk to be fetched in the next interval, followed by video chunks that go back to some low format.

(3) **Look ahead N and look back (LA(N)+LB)**. To overcome the potential smoothness problem of LA(1)+LB, we extend it by allowing the energy surplus/deficit and the N -chunk energy budget to be spread over the next N chunks. Since the basic MPC already looks ahead N chunks in pick-

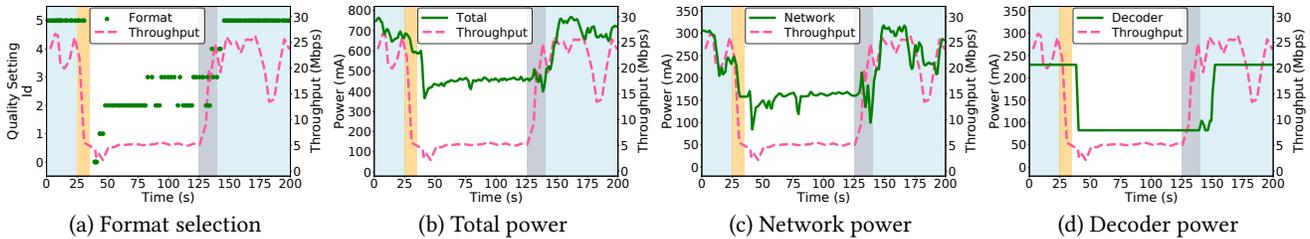


Figure 5: Network throughput, ABR decision, total and individual hardware component power in different network states (**H**: blue; **L**: white; **H-L**: orange; **L-H**: gray). Quality setting: 5: (4K, 60 FPS); 4: (1440p, 60 FPS); 3: (1080p, 60 FPS); 2: (720p, 60 FPS); 1: (480p, 30 FPS); 0: (360p, 30 FPS). The client buffer size is 7s, the same as 4K streaming on Youtube mobile app stated in §9.

ing the next chunk k to optimize the total QoE for them, allowing spreading the energy surplus over the N chunks can be easily incorporated in the modified MPC. In particular, Algorithm 2 adds the total energy $N \cdot P_b \cdot \delta t + E_s$ as the total energy constraint to the dynamic programming which will search among all possible ways to spread the energy surplus among the N intervals to find the one that gives the maximal total QoE for the N intervals. As in the basic MPC, optimizing the total QoE for the next N intervals will take smoothness in the next N chunks into consideration.

7.4 Function-wise Power Prediction

We next describe a practical and accurate function-wise power prediction methodology for use with any proactive energy-aware ABR adaptation such as ENERGY-AWARE ABR.

7.4.1 Why Function-wise Power Prediction?

The obvious choice of using traditional component-wise power models for mobile devices which have been well studied [22, 23, 26, 29, 44, 64, 70, 75, 81, 87, 91] does not work. Such a model derives the correlation between the utilization of each phone component in each of its power states, *e.g.*, utilization and operating frequency, and the resulting power draw using carefully designed microbenchmarks. To use such a model, the hardware component usage is logged *during* app execution and afterwards fed into the power model as input to estimate the component-wise power draw that happened during the app execution. Thus such traditional power models are *postmortem*; they are suitable for post-processing, *e.g.*, monitoring the actual energy drain of a past interval (or calculating the energy surplus/deficit) in reactive or proactive energy-aware app adaptation, but not for predicting the app energy drain in future intervals.

The other design choice is to treat the video streaming app as a black-box and measure and tabulate its average power draw offline when streaming all possible video bitrates under all possible network bandwidth. However, this is not practical as there are potentially infinite number of such combinations. More importantly, such an approach cannot easily model the power draw of *asynchronous component behavior* discussed below where different phone components, *e.g.*, the decoder and the network interface, can be processing different video

chunks (of different bitrates) in a time interval, due to the playback buffer delay effect explained below.

Asynchronous Component Behavior. To illustrate the asynchronous component power behavior in video streaming, we profile the component power draw in an ABR-based 360° video streaming session.

Our experimental setup is the same as in §2 except two differences for enabling power profiling of phone components. First, we build our own ABR server for 360° video streaming with Robust MPC [85] as the default ABR algorithm, and implement our own mobile client using ExoPlayer [3]. Second, we derive the traditional component-wise power model for our experimental phone, Pixel 2, by running microbenchmarks and measuring the phone power draw using an external high-resolution Monsoon power monitor.

Figure 5 shows the profiling result. We see that the network bandwidth went through five stages: high (the H stage), high transitioning to low (the H-L stage), low (the L stage), low transitioning to high (the L-H stage), and finally high again. Figure 5a shows that the network bandwidth change causes the chunk format selected to change almost immediately, *e.g.*, from format 5 during the H stage dropping to format 1 during the H-L stage. However, Figure 5b shows that there is a lag of the total power draw change in following the network bandwidth change, mostly notably at 25-40s and 130-150s. To understand this lag, we zoom into the per-component power draw timeline.

The lag cannot be explained by the CPU and GPU power, both staying almost constant during the session (not shown due to page limit) regardless of the chunk format because of their constant load. The lag also cannot be explained by the network interface (NIC) power which, as shown in Figure 5c, is directly affected by the chunk format and closely follows the format change, *e.g.*, during the H stage and the L stage. In the L-H stage, the NIC power first goes up due to chunk size increase but then goes down as it spends an increasing fraction of the 2-second interval in the idle state as the network bandwidth goes up sharply.

Finally, Figure 5d shows that although in general the hardware decoder power in the H stage is higher than in the L stage because the chunk format and hence decoding load are higher in the H stage, the change of the decoder power

shows a prominent delay behind the network bandwidth change, e.g., when the network bandwidth drops sharply around 24-40s and increases sharply around 130-150s. This explains the lag of the total power draw curve behind the network bandwidth curve. This happens due to the *buffer delay effect* which results in asynchronous power behavior of phone components: while the NIC is downloading the next chunk (e.g., in a low format), the decoder decodes the video chunk at the head of the playback buffer (e.g., in a high format) which was downloaded several intervals ago. The extent of the delay depends on the occupancy of the client buffer as well as network throughput.

7.4.2 Function-wise Power Prediction

The above asynchronous hardware component power behavior suggests that if we cluster the hardware components according to the common video chunk they process at each time interval, the components within each cluster will have synchronous power behavior, i.e., which only depends on the properties of the chunk they process, and such power behavior can be modeled using a power predictor that only uses chunk properties as input. Semantically, each cluster typically corresponds to a high-level app function. We thus propose *function-wise power prediction* that models the power draw of each high-level app function.

In 360° video streaming, there are two primary tasks: (1) *video decoding and displaying* which employs the CPU, GPU, hardware decoder, screen and game rotation vector sensor to process the same chunk in each time interval; (2) *network transmission* for fetching video chunks which involves the CPU and the network interface to process the same chunk (which is different from in task 1) in each interval. In function-wise power prediction, we build the power predictors for these two functions separately.

For the video decoding and displaying function, we make a key observation that the primary hardware components involved are the GPU and hardware decoder, and their power only depend on video properties such as resolution and frame rate but barely depend on video content (based on our measurements). Thus we should be able to develop a power prediction model using these video properties as input. We model the display power separately as a function of the brightness.³ In particular, we measure the total power draw in playing pre-downloaded 360° videos with the same six quality settings as in §2 using the Monsoon power monitor. In offline processing, the video decoding and displaying function power draw is modeled as a piecewise linear function,⁴ $P_{vid} = P(b, res, fps)$, where b , res and fps represent the screen brightness level, video resolution and video frame rate, respectively.

³We did not use possibly more accurate, content-aware OLED power models [25, 28] as OLED display only draws a small amount of energy (§2).

⁴The hardware decoder power draw behaves as a step function of the resolution and FPS.

Algorithm 3: Smoothing in Energy-aware ABR

Input : Selected format FS_{k-1} for interval $k-1$;
 Format F_k for next interval k selected by reactive or proactive energy-aware ABR;
Output : Final format FS_k for next interval k
if $F_k > FS_{k-1}$ **then** $FS_k = FS_{k-1} + 1$;
else $FS_k = F_k$;

For the network transmission function, we develop a linear model that models the power draw as a function of the down-link throughput and chunk size. We experimentally found that running network microbenchmarks does not capture well the share of CPU usage due to downloading when the whole streaming app is running. Instead, we directly stream 360° videos over the Internet, while logging the inputs to both function-wise prediction models, i.e., application events for each 2-second video chunk, including three video properties (resolution, frame rate, and size), start and end time of network transmission, and start and end time of decoding and displaying. We measure the total phone power draw using the power monitor, and then subtract from it the power consumed by the video decoding and displaying function (estimated using its power prediction model built above) as the power for the network transmission function. Finally, the network transmission function power draw is modeled as $P_{net} = P_{\Delta,net} \times \gamma + P_{base,net}$, where γ is the throughput of fetching the video chunk and $P_{\Delta,net}$ and $P_{base,net}$ are derived from linear regression.⁵

8 Adaptation Smoothing

The baseline reactive adaptation Algorithm 1 can result in significant oscillation in selecting the next chunk format, e.g., under high network bandwidth which allows some high format that exceeds the power budget, followed by switching to some low format to compensate for temporary energy deficit (see Figure 3). A proactive adaptation algorithm like Algorithm 2 will not pick arbitrarily high format due to the fixed per-interval energy budget, but exploiting energy surplus from past low-bandwidth intervals can also result in the controller picking some high format transiently since it can only look ahead N chunks. In both cases, the sudden change in format can reduce the smoothness component of the QoE over time (e.g., beyond the N chunk horizon).

To mitigate this potential effect, we propose a smoothing step that can be applied to both reactive and proactive algorithms: it imposes an incremental increase when the chosen format for the next chunk is much higher than the previous one, as shown in Algorithm 3. Note we cannot impose an incremental decrease when newly chosen format is much lower than the previous one, since such a choice is limited by the network bandwidth.

⁵ Network conditions, e.g., signal strength, are reflected in throughput which is one of the model predictors.

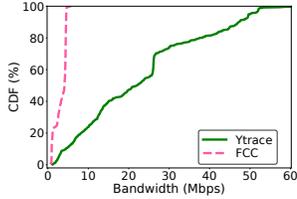


Figure 6: Distribution of network throughput over the traces of two datasets.

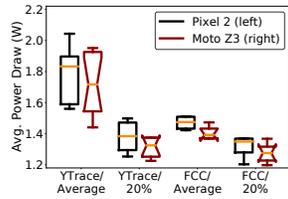


Figure 7: Distribution of average and 20th-percentile per-interval power draw over the traces of two datasets.

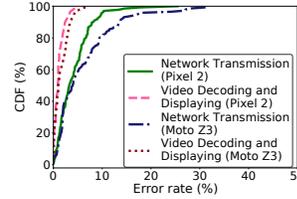


Figure 8: CDF of training accuracy of function-wise power modeling.

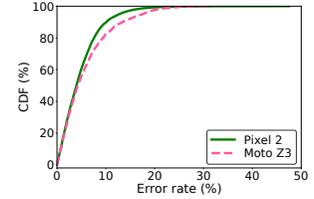


Figure 9: CDF of actual prediction accuracy on Pixel 2 and Moto Z3.

9 Implementation

We implemented the ENERGY-AWARE ABR server on top of Puffer, an open-sourced platform for video streaming [11] in about 1800 lines of C++ code, and built a simple ENERGY-AWARE ABR client that enables 360° video streaming on top of Exoplayer [3] by adding 1.2 KLOC in Java. To save energy on mobile client devices, we implemented both reactive and proactive ENERGY-AWARE ABR on the server side (following [11]). For convenience, we used function-wise power prediction to implement the energy profiler module (using actual throughput) in both types of adaptation schemes and the power predictor module (using predicted throughput) in proactive schemes.

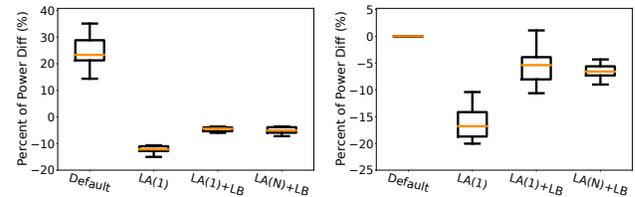
In proactive schemes, the ENERGY-AWARE ABR client checks its buffer occupancy every 0.25 seconds. If it is below the buffer threshold, it reports the current buffer occupancy back to the server. The server then runs the ENERGY-AWARE ABR algorithm to predict the format for the next video chunk, and transmits the video chunk with the selected format to the client. We choose the buffer threshold of 7 seconds based on the observation (using the Youtube built-in tool *stats-for-nerds* [14]) that the Youtube mobile app on the Pixel 2 phone, when streaming 4K 360° videos, requests for the next video chunk when the client-side buffer size is below 7 seconds.

10 Evaluation

In this section, we evaluate the end-to-end performance of reactive and proactive energy-aware ABR streaming players for 360° videos. Our evaluation seeks to answer the following questions: (1) How effective is incorporating energy surplus/deficit in proactive app adaptation? (2) How much does proactive energy-aware adaptation mitigate oscillation and improve QoE compared to reactive approaches? (3) How effective is adaptation smoothing?

10.1 Experimental Setup

We evaluate ENERGY-AWARE ABR performance by streaming videos from a media server to a mobile client, while varying the network condition using network traces from two datasets, YTrace and FCC. We collected YTrace by logging the 2-second average throughput of real users watching 360°



(a) Low power budget

(b) High power budget

Figure 10: Percentage difference between the average power consumption of each streaming session and corresponding power budget for proactive approaches on Pixel 2.

videos on Youtube on mobile devices over around 43200 seconds. FCC [4] is a broadband dataset that has been used in many recent ABR work [15, 59]. Figure 6 shows the distribution of the network throughput of the traces in the two datasets; the average throughput across the traces in the two datasets are 22.89 Mbps and 3.24 Mbps, respectively.

To compare the performance of different ENERGY-AWARE ABR designs under the same network condition, we use the Linux *tc* tool to throttle the throughput along with an 80ms RTT between the server and the client. The video hosting server runs on an Intel i5 2.5GHz processor and runs Ubuntu 18.04. The mobile client streams videos over 802.11n on the Pixel 2/Moto Z3 phones which are connected to a Monsoon power monitor to measure the total phone power draw as the ground truth. Each streaming session lasts for 5 minutes.

The 360° videos chosen from Youtube are characterized into the same three groups as in §2. Each video is segmented into 2-second chunks and encoded into the same six quality settings as in §2. We calculate each encoded video chunk’s SSIM [77] relative to the canonical source as the quality Q_k of the chunk used in the QoE function (Eqn. 1). As the default QoE function, we use the weights $\lambda = 5, \mu = 20$. We also run sensitivity experiments that vary the QoE weights.

We evaluate various streaming approaches under two average power budgets selected as follows. We first stream the 360° videos with the default ABR algorithm without any power constraint, and measure the average power draw of each streaming session and the per-interval average power draw within the same streaming session. Then, for each network trace, we select the 20th-percentile per-interval average power draw as its *low power budget* and the average power draw over the streaming session as its *high power budget* in all our experiments. Figure 7 shows the distribution of the

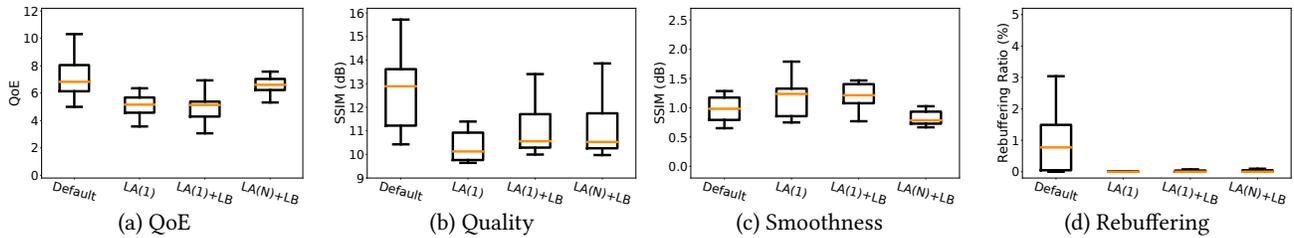


Figure 11: QoE and breakdowns of proactive ENERGY-AWARE ABR under the low power budget on Pixel 2.

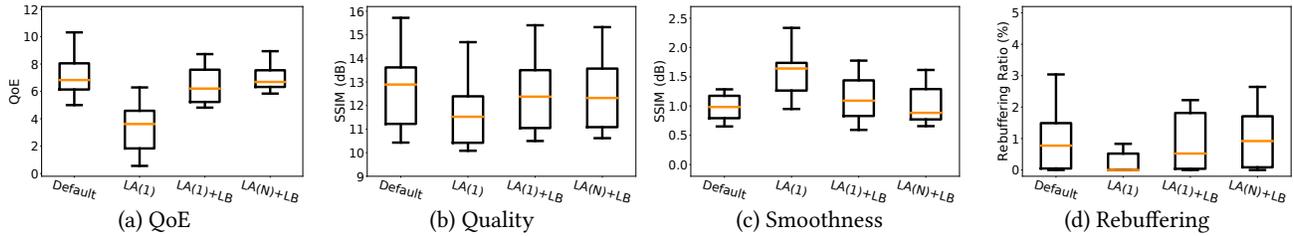


Figure 12: QoE and breakdowns of proactive ENERGY-AWARE ABR under the high power budget on Pixel 2.

two power budgets chosen this way; they vary across the traces of the two data sets. We choose the high power budget in this way to assess the potential penalty of being energy-aware by calculating the performance difference between ENERGY-AWARE ABR and the default ABR.

10.2 Accuracy of Function-wise Power Modeling

We first evaluate the training accuracy of function-wise power modeling for the two functions separately. To train the model for the video decoding and displaying function, we randomly select one video from every video group and play it locally on the Pixel 2 and Moto Z3 phones, respectively. To train the model for the network transmission function, we randomly select 10% network traces from each network dataset and we stream one 360° video for 1.2 hours in total. Figure 8 shows that the average error rate of per-interval power draw in training for the two functions are 4.21% and 1.16% on Pixel 2 and 6.11% and 1.65% on Moto Z3, respectively.

We next validate our power model by comparing the estimated average per-interval power consumption of 360° video streaming against the power monitor readings over 50% of the remaining network traces. Figure 9 shows that function-wise power predictor achieves mean estimation accuracy of 4.87% and 5.86% in estimating the per-interval average power consumption on Pixel 2 and Moto Z3 phones, respectively.

10.3 Proactive Energy-aware ABR

We first evaluate the three designs of proactive ENERGY-AWARE ABR. We focus on Pixel 2; the results for Moto Z3 are very similar and are omitted due to page limit.

Power consumption. Figure 10a shows that for the low power budget, all three proactive designs consume less power than the power budget. The average power consumption for LA(1)+LB and LA(N)+LB are only 4.09% and 4.80% below the given power budget, respectively, suggesting both designs

Table 2: Comparison between reactive and proactive ENERGY-AWARE ABR without and with smoothing under low power budget on Pixel 2 and Moto Z3 (average/standard deviation).

	RA	RA+S	LA(N)+LB	LA(N)+LB+S
Pixel 2				
Power Diff (%)	-3.50/0.96	-3.40/1.43	-4.80/1.90	-3.78/1.61
QoE	4.02/1.31	4.91/0.39	6.74/0.65	7.11/0.60
Quality (dB)	11.19/1.05	11.33/1.09	11.57/1.23	11.59/1.23
Smoothness (dB)	1.43/0.27	1.27/0.20	0.96/0.30	0.90/0.25
Rebuffering (%)	0.00/0.00	0.12/0.29	0.02/0.05	0.00/0.00
Moto Z3				
Power Diff (%)	0.42/1.53	0.24/1.94	-1.73/1.83	-0.59/1.56
QoE	4.27/1.03	5.36/0.81	5.79/0.97	6.39/0.81
Quality (dB)	11.46/0.52	11.52/0.42	11.50/0.55	11.55/0.52
Smoothness (dB)	1.47/0.28	1.21/0.22	1.13/0.27	1.03/0.18
Rebuffering (%)	0.14/0.32	0.22/0.48	0.13/0.28	0.03/0.06

efficiently exploit the energy saved during past intervals in downloading future video chunks; the small gap to the given power budget comes from discretized chunk formats. In contrast, LA(1) significantly under-utilizes the power budget by 12.20% on average, from not exploiting energy surplus accumulated from low network bandwidth intervals.

Figure 10b shows that for the high power budget, the trend is similar; LA(1)+LB and LA(N)+LB only under-utilize the power budget by 5.65% and 6.58%, while LA(1) under-utilizes by 15.39%. The larger gaps compared to the low power budget scenario are because all schemes are juggling among higher chunk formats due to the high power budget, which have larger discretization effects of video encoding.

User experience. We next compare the user experience of the three designs under the two power budgets. Figure 11a shows that for the low power budget, LA(N)+LB achieves the highest average QoE of 6.61, compared with 4.75 for LA(1) and 4.83 for LA(1)+LB.

To understand how different designs affect the QoE for the low power budget, we break down QoE into its three components: video quality, smoothness and rebuffering (Eqn. 1). (1)

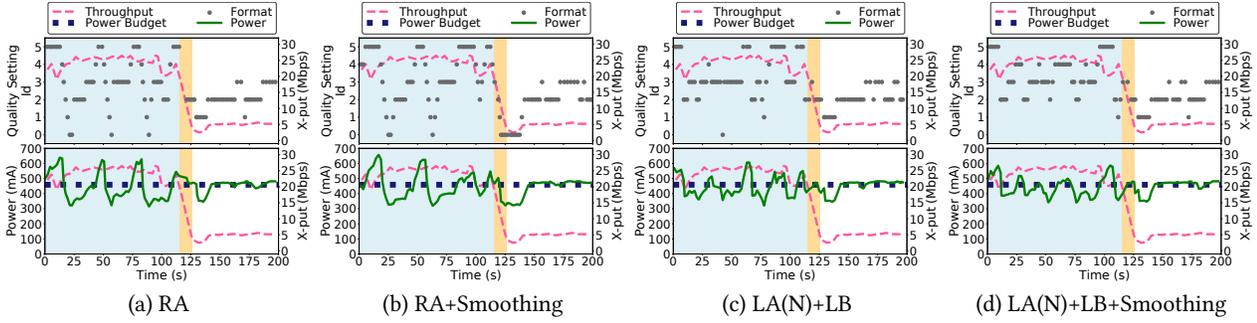


Figure 13: Low power budget case study: format selection and power behavior of reactive and proactive designs with and without smoothing under a sample network trace on Pixel 2.

Figure 11b shows that LA(1)+LB and LA(N)+LB have similar video quality, 11.18 dB and 11.14 dB, respectively, both higher than the quality of LA(1) of 10.57 dB. It suggests that exploiting energy surplus saved during past intervals in LA(1)+LB and LA(N)+LB improves the video quality of future intervals. (2) Figure 11c shows that LA(N)+LB has the smallest mean quality change of 0.89 dB, compared with 1.15 dB for LA(1) and 1.26 dB for LA(1)+LB. It is the same as that of the default ABR control. It suggests that looking ahead the power consumption of future N intervals effectively smooths the quality switching. (3) Figure 11d shows that all three energy-aware designs have similarly low average rebuffering ratio of around 0.14%, since the low power budget leads to lower chunk formats selected by the ENERGY-AWARE ABR controller which reduce the rebuffering time for all designs.

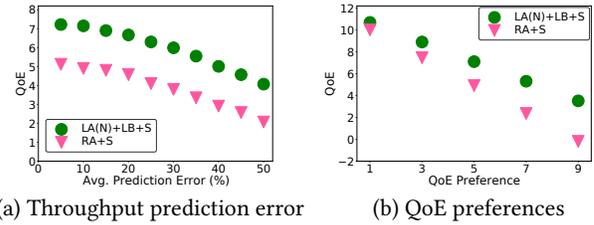
Figure 12 shows that the high power budget scenario has similar user experience results as the low power budget scenario, where LA(N)+LB achieves the highest average QoE of 6.94 and the closest gap with the default ABR of only 4.1%. The slightly low QoE comes from 0.28 dB video quality reduction (2.2%) as shown in Figure 12b. This shows the penalty of proactive energy-aware adaptation compared to the energy-oblivious default ABR is really small.

10.4 Reactive vs. Proactive Energy-aware ABR

We next evaluate the benefits of proactive energy-aware app adaptation by comparing reactive and proactive ENERGY-AWARE ABR, with and without adaptation smoothing, under the low power budget, on Pixel 2 and Moto Z3 phones.

Power consumption. Table 2 shows that all four designs satisfy the power budget with a small gap of 4.80%–3.40% below it on Pixel 2 and -1.73%–0.42% on Moto Z3. The small gap can be explained by the small discretization effects among the low chunk formats selected in the low budget scenario.

User experience. Table 2 shows when the power budget is lower than the default app energy power draw, proactive energy-aware app adaptation shows significant benefits over reactive adaptation on both phones. We next elaborate on the results on Pixel 2. (1) Without smoothing, LA(N)+LB achieves much higher (67.7%) mean QoE than RA, 6.74 over



(a) Throughput prediction error (b) QoE preferences

Figure 14: Sensitivity analysis under low power budget on Pixel 2.

4.02. As expected, the improvement mainly comes from significantly improved smoothness, 0.96 dB for LA(N)+LB and 1.43 dB for RA. (2) Smoothing improves QoE for both reactive and proactive design, by 0.89 and 0.37, respectively, primarily from improved smoothness of 0.16 dB for RA and 0.06 dB for LA(N)+LB. (3) With smoothing, LA(N)+LB+Smoothing achieves 44.8% higher mean QoE than RA+Smoothing, 7.11 over 4.91. The improvement mainly comes from significantly reduced quality switching (0.37 dB reduction) and significantly lower rebuffering ratio (0.12% reduction) and to a small extent the 0.26 dB higher quality. Table 2 also shows LA(N)+LB+Smoothing achieves 19.2% higher mean QoE than RA+Smoothing, 6.39 over 5.36, on Moto Z3.

Case study. Figure 13 shows a case study on Pixel 2 to explain how proactive designs reduce format oscillation compared to reactive designs. The streaming session was under the low power budget, and the network bandwidth went through 3 stages: high (H), high-to-low (H-L), and low (L).

The comparison at the H stage shows that proactive designs can reduce the oscillation when the network bandwidth is high. RA adjusts the format without knowing how much to adapt, which leads to frequent format oscillation to correct energy drain deviation as seen around 20–30s, 50–70s, 80–100s. Instead of aggressively increasing the format when there is no energy deficit, RA+Smoothing increases the format by 1 at each step at around 30–50s and 70–90s. In contrast, LA(N)+LB with and without smoothing do not rapidly go up and down all formats and stay in each format longer, from incorporating the power budget in selecting chunk formats. While LA(N)+LB occasionally jumps formats, e.g., to format 5 from format 3 around 60–70s and 105–115s, LA(N)+LB+Smoothing

further improves smoothness by gradually increasing the format when there is energy surplus, *e.g.*, from format 3 to format 4 around 60-70s and 90-100s,

10.5 Sensitivity Analysis

Throughput prediction accuracy. To study the impact of network throughput prediction error on QoE, we evaluate reactive and proactive designs with smoothing under the low power budget by modeling the throughput prediction as a combination of the ground truth throughput with random noise according to the given average error level. Figure 14a shows that the throughput prediction error influences both reactive and proactive approaches, but the gaps remain similar. In particular, LA(N)+LB+S performs better than RA+S by 41.0%-96.6% for the low power budget on Pixel 2.

User QoE preference. We compare QoE of RA+S and LA(N)+LB+S under 5 different QoE weights for smoothness, {1, 3, 5, 7, 9}, while keeping the weights for quality and re-buffering at 1 and 20, respectively. Figure 14b shows that as users put more penalty weight of smoothness, the difference between QoE of LA(N)+LB+S and RA+S increases from 0.68 (1.07X) to 3.69 (19.4X) for the low power budget on Pixel 2.

11 Discussion

Multiple apps competing for the energy budget. In this paper, we focused on energy-aware adaptation of a single app, *e.g.*, one that dominates the phone energy drain in a four-hour plane ride. In practice, the user may be switching among multiple apps, and there could be unexpected background apps that also drain energy from the total energy budget. In the first case, the integrated proactive app adaptation can be applied to each app while it is running, *e.g.*, constrained by the average power budget for the plane ride. If several apps run concurrently, *e.g.*, some in background and some in foreground, the user can potentially provide input on how the total energy/power budget should be split among the apps. Alternatively, a global energy-aware controller could be developed, *e.g.*, in the OS, to jointly optimize the QoE of concurrently running apps while satisfying the total energy/power budget.

Leveraging hysteresis in proactive adaptation. Reactive adaptation (*e.g.*, [31]) mitigates the oscillation problem by leveraging hysteresis, *i.e.*, imposing a threshold that the energy surplus/deficit must exceed in order to trigger fidelity adaptation (§6). In contrast, our proactive adaptation design, LA(N)+LB, already alleviates oscillation by allowing the energy surplus/deficit and the N -chunk energy budget to be spread over the next N chunks in maximizing the QoE. Incorporating hysteresis into proactive adaptation designs, *e.g.*, leveraging the energy surplus/deficit (while looking ahead N chunks) only when exceeding some threshold, may result

in further smoothness but also lower video quality because of its conservativeness.

12 Related Work

We already discussed previous work on reactive energy-aware app adaptation in §3. Below, we discuss related work on normal and 360° video streaming.

Energy measurements and optimization of video streaming. Many works [41, 79, 89] measure the energy consumption of commercial regular video streaming services in WiFi and LTE networks. Recent studies focus on power measurement of 360° video streaming [48, 87]. Many works [16, 20, 24, 27, 35, 40, 43, 53, 55, 56, 58, 60, 78, 90] propose techniques to minimize energy drain for video streaming via bandwidth control, packet scheduling, screen brightness scaling, *etc.* RnB [84] studies the problem of jointly adapting video bitrate and display brightness to reduce energy consumption while maintaining a quality goal. These works are orthogonal to our work; they focus on energy drain measurement or optimization, while our work focuses on energy-aware adaptation, *i.e.*, how to maximize QoE while satisfying user-configurable power constraint.

360° video streaming. Many works study supporting 360° video streaming on head-mounted displays or commodity phones. Several works [18, 50] propose to pre-cache panoramic frames to provide the clients the freedom of changing orientation during playback. Other works [39, 42, 65, 69, 80, 93] exploit different projection and tile-based or viewport-based video encoding schemes to save network bandwidth. They perform network-aware adaptation rather than energy-aware adaptation. We did not evaluate viewport-based 360° video streaming because commercial video streaming like Youtube does not use it and viewport prediction in the 2-second scale has been shown to be inaccurate; the median longitude error is about 20 degrees [82].

13 Conclusion

In this paper, using 360° video streaming as a case study, we showed that proactive energy-aware app adaptation that integrates the user-specified energy drain budget into the QoE optimizer of modern apps can significantly reduce app fidelity oscillation and improve the app QoE over traditional reactive energy-aware app adaptation. We believe that proactive energy-aware app adaptation is rather general and as future work, we will validate its applicability and effectiveness for other power-hungry modern apps with built-in adaptation logic such as the class of mobile apps that adaptively offload computation to edge servers.

Acknowledgement We thank our shepherd Amy Lynn Murphy and the anonymous reviewers for their helpful comments. This work was supported in part by NSF/Intel grant 1719369.

References

- [1] Planets. <https://www.youtube.com/watch?v=qhLExhpXX0E>, 2017.
- [2] Monster. <https://www.youtube.com/watch?v=a9nV5JvM9Tw>, 2018.
- [3] Exoplayer. <https://exoplayer.dev/>, 2019.
- [4] Federal communications commission. measuring broadband america. <https://www.fcc.gov/general/measuring-broadband-america>, 2019.
- [5] Orleans. https://www.youtube.com/watch?v=bSV8qc2_qFs, 2019.
- [6] Solar. https://www.youtube.com/watch?v=MnmU_NwhTFU, 2019.
- [7] Top mobile app development trends in 2020. <https://www.smartinsights.com/mobile-marketing/app-marketing/top-mobile-app-development-trends-in-2020-infographic/>, 2019.
- [8] Battery draining quickly? find out which apps are to blame. <https://www.komando.com/smartphones-gadgets/battery-draining-quickly-find-out-which-apps-are-to-blame/698674/>, 2020.
- [9] Chicago. <https://www.youtube.com/watch?v=Gu1D3BnIYZg>, 2020.
- [10] Optical. https://www.youtube.com/watch?v=x_rN5YUXZi8, 2020.
- [11] Puffer. <https://puffer.stanford.edu/>, 2020.
- [12] Techengage. top 10 battery draining apps to avoid 2020. <https://techengage.com/top-battery-draining-apps-to-avoid/>, 2020.
- [13] Techrepublic. the most battery-draining apps of 2020. <https://www.techrepublic.com/article/the-most-battery-draining-apps-of-2020/>, 2020.
- [14] Youtube stats-for-nerds. <https://support.google.com/youtube/thread/3284269?hl=en>, 2020.
- [15] Z. Akhtar, Y. S. Nam, R. Govindan, S. Rao, J. Chen, E. Katz-Bassett, B. Ribeiro, J. Zhan, and H. Zhang. Oboe: auto-tuning video abr algorithms to network conditions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 44–58, 2018.
- [16] F. Albiero, J. Vehkaperä, M. Katz, and F. Fitzek. Overall performance assessment of energy-aware cooperative techniques exploiting multiple description and scalable video coding schemes. In *6th Annual Communication Networks and Services Research Conference (CNSR 2008)*, pages 18–24. IEEE, 2008.
- [17] A. Bemporad and M. Morari. Robust model predictive control: A survey. In *Robustness in identification and control*, pages 207–226. Springer, 1999.
- [18] M. Berning, T. Yonezawa, T. Riedel, J. Nakazawa, M. Beigl, and H. Tokuda. panorama: 360 degree interactive video for augmented reality prototyping. In *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication, UbiComp '13 Adjunct*, pages 1471–1474, New York, NY, USA, 2013. ACM.
- [19] E. F. Camacho and C. B. Alba. *Model predictive control*. Springer Science & Business Media, 2013.
- [20] N. Chang, I. Choi, and H. Shim. Dls: dynamic backlight luminance scaling of liquid crystal display. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(8):837–846, 2004.
- [21] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 155–168, 2015.
- [22] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone energy drain in the wild: Analysis and implications. *ACM SIGMETRICS Performance Evaluation Review*, 43(1):151–164, 2015.
- [23] X. Chen, J. Meng, Y. C. Hu, M. Gupta, R. Hasholzner, V. N. Ekambaram, A. Singh, and S. Srikanteswara. A fine-grained event-based modem power model for enabling in-depth modem energy drain analysis. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(2):1–28, 2017.
- [24] L. Cheng, S. Mohapatra, M. El Zarki, N. Dutt, and N. Venkatasubramanian. Quality-based backlight optimization for video playback on handheld devices. *Advances in Multimedia*, 2007, 2007.
- [25] P. Dash and Y. C. Hu. How much battery does dark mode save? an accurate oled display power profiler for modern smartphones. In *Proceedings of ACM MobiSys*, 2021.
- [26] N. Ding and Y. C. Hu. Gfxdoctor: A holistic graphics energy profiler for mobile devices. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 359–373, 2017.

- [27] F. R. Dogar, P. Steenkiste, and K. Papagiannaki. Catnap: exploiting high bandwidth wireless interfaces to save energy for mobile devices. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 107–122, 2010.
- [28] M. Dong, Y.-S. K. Choi, and L. Zhong. Power modeling of graphical user interfaces on oled displays. In *Proceedings of the 46th Annual Design Automation Conference*, pages 652–657. ACM, 2009.
- [29] M. Dong and L. Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 335–348, 2011.
- [30] A. E. Eshratifar, M. S. Abrishami, and M. Pedram. Jointdnn: an efficient training and inference engine for intelligent mobile cloud computing services. *IEEE Transactions on Mobile Computing*, 2019.
- [31] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. *ACM SIGOPS Operating Systems Review*, 33(5):48–63, 1999.
- [32] J. Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Proceedings WMCSA'99. Second IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–10. IEEE, 1999.
- [33] R. Fonseca, P. Dutta, P. Levis, and I. Stoica. Quanto: Tracking energy in networked embedded systems. In *OSDI*, volume 8, pages 323–338, 2008.
- [34] C. E. Garcia, D. M. Prett, and M. Morari. Model predictive control: theory and practice—a survey. *Automatica*, 25(3):335–348, 1989.
- [35] Y. Go, O. C. Kwon, and H. Song. An energy-efficient http adaptive video streaming with networking cost constraint over heterogeneous wireless networks. *IEEE Transactions on Multimedia*, 17(9):1646–1657, 2015.
- [36] L. Guo, T. Xu, M. Xu, X. Liu, and F. X. Lin. Power sandbox: Power awareness redefined. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.
- [37] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 68–81. ACM, 2014.
- [38] J. He, M. A. Qureshi, L. Qiu, J. Li, F. Li, and L. Han. Rubiks: Practical 360-degree streaming for smartphones. In *MobiSys*, 2018.
- [39] J. He, M. A. Qureshi, L. Qiu, J. Li, F. Li, and L. Han. Rubiks: Practical 360-degree streaming for smartphones. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '18*, pages 482–494, New York, NY, USA, 2018. ACM.
- [40] M. A. Hoque, M. Siekkinen, and J. K. Nurminen. Using crowd-sourced viewing statistics to save energy in wireless video streaming. In *Proceedings of the 19th annual international conference on Mobile computing & networking*, pages 377–388, 2013.
- [41] M. A. Hoque, M. Siekkinen, J. K. Nurminen, and M. Aalto. Dissecting mobile video services: An energy consumption perspective. In *2013 IEEE 14th International Symposium on "A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*, pages 1–11. IEEE, 2013.
- [42] M. Hosseini and V. Swaminathan. Adaptive 360 VR video streaming: Divide and conquer! *CoRR*, abs/1609.08729, 2016.
- [43] W. Hu and G. Cao. Energy-aware video streaming on smartphones. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1185–1193. IEEE, 2015.
- [44] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *Mobisys*, 2012.
- [45] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 187–198, 2014.
- [46] J. Jiang, V. Sekar, and H. Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 97–108, 2012.
- [47] N. Jiang, Y. Liu, T. Guo, W. Xu, V. Swaminathan, L. Xu, and S. Wei. Qurate: power-efficient mobile immersive video streaming. In *Proceedings of the 11th ACM Multimedia Systems Conference*, pages 99–111, 2020.
- [48] N. Jiang, V. Swaminathan, and S. Wei. Power evaluation of 360 vr video streaming on head mounted display devices. In *Proceedings of the 27th Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 55–60, 2017.

- [49] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News*, 45(1):615–629, 2017.
- [50] E. Kuzyakov and D. Pio. Next-generation video encoding techniques for 360 video and vr.(2016). <https://code.facebook.com/posts/1126354007399553/nextgeneration-video-encoding-techniques-for-360-video-and-vr>, 2016.
- [51] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane. Spinn: synergistic progressive inference of neural networks over device and cloud. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–15, 2020.
- [52] E. Li, L. Zeng, Z. Zhou, and X. Chen. Edge ai: On-demand accelerating deep neural network inference via edge computing. *IEEE Transactions on Wireless Communications*, 19(1):447–457, 2019.
- [53] X. Li, M. Dong, Z. Ma, and F. C. Fernandes. Greentube: power optimization for mobile videostreaming via dynamic cache management. In *Proceedings of the 20th ACM international conference on Multimedia*, pages 279–288, 2012.
- [54] Z. Li, X. Zhu, J. Gahm, R. Pan, H. Hu, A. C. Begen, and D. Oran. Probe and adapt: Rate adaptation for http video streaming at scale. *IEEE Journal on Selected Areas in Communications*, 32(4):719–733, 2014.
- [55] C.-H. Lin, P.-C. Hsiu, and C.-K. Hsieh. Dynamic backlight scaling optimization: A cloud-based energy-saving service for mobile streaming applications. *IEEE Transactions on Computers*, 63(2):335–348, 2012.
- [56] J. Liu and L. Zhong. Micro power management of active 802.11 interfaces. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, pages 146–159, 2008.
- [57] L. Liu, H. Li, and M. Gruteser. Edge assisted real-time object detection for mobile augmented reality. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.
- [58] Y. Liu, M. Xiao, M. Zhang, X. Li, M. Dong, Z. Ma, Z. Li, and S. Chen. Gocad: Gpu-assisted online content-adaptive display power saving for mobile devices in internet streaming. In *Proceedings of the 25th International Conference on World Wide Web*, pages 1329–1338, 2016.
- [59] H. Mao, R. Netravali, and M. Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210, 2017.
- [60] S. Mohapatra, R. Cornea, H. Oh, K. Lee, M. Kim, N. Dutt, R. Gupta, A. Nicolau, S. Shukla, and N. Venkatasubramanian. A cross-layer approach for power-performance optimization in distributed mobile systems. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 8–pp. IEEE, 2005.
- [61] R. K. Mok, X. Luo, E. W. Chan, and R. K. Chang. Qdash: a qoe-aware dash system. In *Proceedings of the 3rd Multimedia Systems Conference*, pages 11–22, 2012.
- [62] R. Neugebauer and D. McAuley. Energy is just another resource: Energy accounting and energy pricing in the nemesis os. In *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, pages 67–72. IEEE, 2001.
- [63] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 29–42, 2012.
- [64] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the sixth conference on Computer systems*, pages 153–168, 2011.
- [65] F. Qian, B. Han, Q. Xiao, and V. Gopalakrishnan. Flare: Practical viewport-adaptive 360-degree video streaming for mobile devices. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking, MobiCom '18*, pages 99–114, New York, NY, USA, 2018. ACM.
- [66] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen. Deepdecision: A mobile deep learning framework for edge video analytics. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 1421–1429. IEEE, 2018.
- [67] J. B. Rawlings and D. Q. Mayne. *Model predictive control: Theory and design*. Nob Hill Pub., 2009.
- [68] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. Energy management in mobile devices with the cinder operating system. In *Proceedings of the sixth conference on Computer systems*, pages 139–152, 2011.
- [69] S. Shi, V. Gupta, and R. Jana. Freedom: Fast recovery enhanced vr delivery over mobile networks. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '19*, pages 130–141, New York, NY, USA, 2019. ACM.

- [70] A. Shye, B. Scholbrock, and G. Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 168–178, 2009.
- [71] I. Sodagar. The mpeg-dash standard for multimedia streaming over the internet. *IEEE multimedia*, 18(4):62–67, 2011.
- [72] K. Spiteri, R. Sitaraman, and D. Sparacio. From theory to practice: Improving bitrate adaptation in the dash reference player. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 15(2s):1–29, 2019.
- [73] K. Spiteri, R. Urgaonkar, and R. K. Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.
- [74] L. Sun, Y. Mao, T. Zong, Y. Liu, and Y. Wang. Flocking-based live streaming of 360-degree video. In *Proceedings of the 11th ACM Multimedia Systems Conference*, pages 26–37, 2020.
- [75] L. Sun, R. K. Sheshadri, W. Zheng, and D. Koutsonikolas. Modeling wifi active power/energy consumption in smartphones. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 41–51. IEEE, 2014.
- [76] Y. Sun, X. Yin, J. Jiang, V. Sekar, F. Lin, N. Wang, T. Liu, and B. Sinopoli. Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 272–285, 2016.
- [77] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.
- [78] S. Wei, V. Swaminathan, and M. Xiao. Power efficient mobile video streaming using http/2 server push. In *2015 IEEE 17th International Workshop on Multimedia Signal Processing (MMSP)*, pages 1–6. IEEE, 2015.
- [79] Y. Xiao, R. S. Kalyanaraman, and A. Yla-Jaaski. Energy consumption of mobile youtube: Quantitative measurement and analysis. In *2008 The Second International Conference on Next Generation Mobile Applications, Services, and Technologies*, pages 61–69. IEEE, 2008.
- [80] X. Xie and X. Zhang. Poi360: Panoramic mobile video telephony over lte cellular networks. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies, CoNEXT '17*, pages 336–349, New York, NY, USA, 2017. ACM.
- [81] F. Xu, Y. Liu, Q. Li, and Y. Zhang. V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 43–55, Lombard, IL, 2013. USENIX.
- [82] T. Xu, B. Han, and F. Qian. Analyzing viewport prediction under different vr interactions. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 165–171, 2019.
- [83] F. Y. Yan, H. Ayers, C. Zhu, S. Fouladi, J. Hong, K. Zhang, P. Levis, and K. Winstein. Learning in situ: a randomized experiment in video streaming. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 495–511, 2020.
- [84] Z. Yan and C. W. Chen. Rnb: Rate and brightness adaptation for rate-distortion-energy tradeoff in http adaptive streaming over mobile devices. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 308–319, 2016.
- [85] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 325–338, 2015.
- [86] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 387–400, 2012.
- [87] C. Yue, S. Sen, B. Wang, Y. Qin, and F. Qian. Energy considerations for abr video streaming to smartphones: measurements, models and insights. In *Proceedings of the 11th ACM Multimedia Systems Conference*, pages 153–165, 2020.
- [88] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Ecosystem: Managing energy as a first class operating system resource. *ACM SIGOPS operating systems review*, 36(5):123–132, 2002.
- [89] J. Zhang, G. Fang, C. Peng, M. Guo, S. Wei, and V. Swaminathan. Profiling energy consumption of dash video streaming over 4g lte networks. In *Proceedings of the 8th International Workshop on Mobile Video*, pages 1–6, 2016.

- [90] J. Zhang, Z.-J. Wang, Z. Quan, J. Yin, Y. Chen, and M. Guo. Optimizing power consumption of mobile devices for video streaming over 4g lte networks. *Peer-to-Peer Networking and Applications*, 11(5):1101–1114, 2018.
- [91] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 105–114, 2010.
- [92] Z. Zhao, K. M. Barijough, and A. Gerstlauer. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, 2018.
- [93] C. Zhou, Z. Li, and Y. Liu. A measurement study of Oculus 360 degree video streaming. In *Proceedings of the 8th ACM on Multimedia Systems Conference, MMSys'17*, pages 27–37. ACM, 2017.

Video Analytics with Zero-streaming Cameras

Mengwei Xu^{1,2*}, Tiantu Xu^{3*}, Yunxin Liu⁴, and Felix Xiaozhu Lin⁵

¹Peking University

²Beijing University of Posts and Telecommunications

³Purdue ECE

⁴Institute for AI Industry Research (AIR), Tsinghua University

⁵University of Virginia

Abstract

Low-cost cameras enable powerful analytics. An unexploited opportunity is that most captured videos remain “cold” without being queried. For efficiency, we advocate for these cameras to be *zero streaming*: capturing videos to local storage and communicating with the cloud only when analytics is requested.

How to query zero-streaming cameras efficiently? Our response is a camera/cloud runtime system called DIVA. It addresses two key challenges: to best use limited camera resource during video capture; to rapidly explore massive videos during query execution. DIVA contributes two unconventional techniques. (1) When capturing videos, a camera builds sparse yet accurate landmark frames, from which it learns reliable knowledge for accelerating future queries. (2) When executing a query, a camera processes frames in multiple passes with increasingly more expensive operators. As such, DIVA presents and keeps refining inexact query results throughout the query’s execution. On diverse queries over 15 videos lasting 720 hours in total, DIVA runs at more than 100× video realtime and outperforms competitive alternative designs. To our knowledge, DIVA is the first system for querying large videos stored on low-cost remote cameras.

1 Introduction

Cameras are pervasive: a survey of 61 organizations shows that from 2015 to 2018 their average number of cameras has increased by almost 70%, from 2,900 to 4,900 [6]. Insights of videos can be extracted by queries such as “get the daily peak pedestrian count in the past week” [36, 67, 82, 101]. Four recent trends motivate our work.

*Mengwei Xu and Tiantu Xu contributed equally to the paper.

*Work done during Mengwei Xu’s visit to Purdue University.

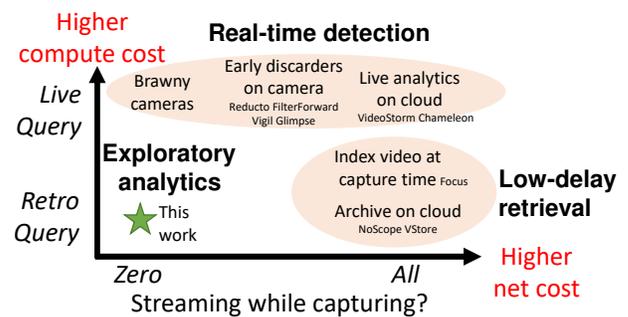


Figure 1: **The design space** of video analytics systems, showing this work and prior systems.

(1) *Low-cost, wireless cameras grow fast* As key complements to high-end cameras, low-cost cameras (<\$40) are increasingly pervasive [17, 18, 34]. These cameras often have limited compute resources yet spacious storage. Being wireless, these cameras are meant to be installed by individuals or small businesses with ease just as other wireless sensors.

(2) *Most videos are cold* Users deploy cameras to knowingly capture excessive videos, expecting that most videos will never be queried [72]. This is because interesting events are often unforeseeable, e.g., car accidents; the need for examining such events emerges well after the fact. §2.1 presents a 6-month study of real-world camera deployment, where only <0.005% of captured videos are eventually queried.

(3) *Transmitting cold videos wastes wireless bandwidth* Cold videos should not compete with human users for network bandwidth. If streaming video in real-time, a single camera generates traffic at 0.2 MB/s–0.4 MB/s (720P@1–30 FPS); with multiple cameras on one network, their always-on streams easily consume most, if not all, bandwidth of consumer WiFi, which is 0.2 MB/s–3 MB/s (median: 0.99) in a recent global survey [9] and less than 1.5 MB/s in an academic study [47]. A dedicated network for cameras is expensive, as the network monetary cost will exceed the camera cost in several months [14].

(4) *Camera storage can retain videos long enough* A cheap

camera can already store videos for weeks or months. Such retention periods already satisfy many video scenarios [2, 10]. In fact, legal regulations often *prevent* retention longer than a few months, mandating video deletion for privacy [1, 7]. Existing measures can assure data security of on-camera videos. §2.3 will provide evidence in detail.

Zero streaming & its use cases How to analyze cold videos produced by numerous low-cost cameras? We advocate for a system model dubbed “zero streaming”. (1) Cameras continuously capture videos to their local storage without uploading any. (2) Only in response to a retrospective query, the cloud reaches out to the queried camera and coordinates with it to process the queried video. (3) While the video is being processed, the system presents users with inexact yet useful results; it continuously refines the results until query completion [50]. In this way, a user may *explore* videos through interactive queries, e.g., aborting an ongoing query based on inexact results and issuing a new query with revised parameters [45, 46]. Zero streaming has rich use cases, for example:

- To trace the cause of recent frequent congestion on a highway, a city planner queries cameras on nearby local roads, requesting car counts seen on these local roads.
- To understand how recent visitors impact bobcat activities, a ranger queries all the park’s cameras, requesting time ranges where the cameras capture bobcats.

Advantages Zero streaming suits resource-frugal cameras in large deployment. When capturing videos, cameras require no network or external compute resources. Only to process a query, the cameras require networks such as long-range wireless [35] and cloud resources such as GPU. Zero streaming adds a new point to the design space of video analytics shown in Figure 1. It facilitates retrospective, exploratory analytics, a key complement to real-time event detection and low-delay video retrieval [51, 55, 65, 99]. The latter demands higher compute or network resources per camera and hence suits fewer cameras around hot locations such as building entrances.

DIVA To support querying zero-streaming cameras, we present a camera/cloud runtime called DIVA. As shown in Figure 2, a camera captures video to local storage; it deletes videos after their maximum retention period. In response to a query, the camera works in conjunction with the cloud: the camera runs operators, implemented as lightweight neural nets (NNs), to *rank* or *filter* frames; the cloud runs full-fledged object detection to validate results uploaded from the camera. DIVA thus does not sacrifice query accuracy, ensuring it as high as that of object detection by the cloud.

The major challenges to DIVA are two. (1) During video capture: how should cameras best use limited resources for future queries? (2) To execute a query: how should the cloud and the camera orchestrate to deliver useful results rapidly? Existing techniques are inadequate. Recent systems pre-process (“index”) video frames as capturing them [51] and answer

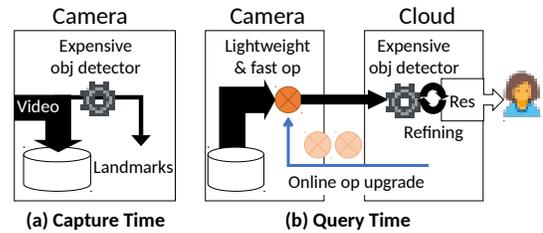


Figure 2: Overview of DIVA

queries based on indexes only. Yet, as we will show in §8, low-cost cameras can hardly build quality indexes in real-time. Many systems process video frames in a streaming fashion [40, 42, 92, 97, 100], which however miss key opportunities in retrospective queries.

To this end, DIVA has two unconventional designs.

• **During video capture: building sparse but sure landmarks to distill long-term knowledge** (Figure 2(a)) To optimize future queries, our key insight is that *accurate* knowledge on a *sparse* sample of frames is much more useful than *inaccurate* knowledge on *all* frames. This is opposite to existing designs that detect objects with low accuracy on all/most frames as capturing them [40, 51]. On a small sample of captured video frames dubbed *landmarks*, the camera runs generic, expensive object detection, e.g., YOLOv3 [77]. Constrained by camera hardware, landmarks are sparse in time, e.g., 1 in every 30 seconds; yet, with high-accuracy object labels, they provide reliable spatial distributions of various objects over long videos. High accuracy is crucial, as we will validate through evaluation (§8.3). DIVA optimizes queries with landmarks: it prioritizes processing of frame regions with object skewness learned from landmarks; it bootstraps operators with landmarks as training samples. Landmarks only capture a small fraction of object instances; those uncaptured do not affect correctness/accuracy (§4).

• **To execute queries: multipass processing with online operator upgrade** (Figure 2(b)) To process large videos, our key insight is to refine query results in multiple passes, each pass with a more expensive/accurate operator. Unlike prior systems processing all frames in one pass and delivering results in one shot [40, 58, 59], multipass processing produces useful results during query execution, enabling users to explore videos effectively. To do so, DIVA’s cloud trains operators with a wide spectrum of accuracies/costs. Throughout query execution, the cloud keeps pushing new operators to the camera, picking the next operator based on query progress, network conditions, and operator accuracy. The early operators quickly *explore* the frames for inexact answers while later operators slowly *exploit* for more exact answers.

On 720-hour videos in total from 15 different scenes, DIVA runs queries at more than 100× video realtime on average, with typical wireless conditions and low-cost hardware. DIVA returns results quickly: compared to executing a query to

completion, DIVA takes one order of magnitude shorter time to return half of the result frames. Compared to competitive alternatives, DIVA speeds up queries by at least 4×.

Contributions We have made the following contributions.

- Zero streaming, a new model for low-cost cameras to operate on frugal networks while answering video queries.
- Two novel techniques for querying zero-streaming cameras: optimizing queries with accurate knowledge from sparse frames; processing frames in multiple passes with operators continuously picked during a query.
- DIVA, a concrete implementation that runs queries at more than 100× realtime with uncompromised query accuracy. To our knowledge, DIVA is the first system designed for querying large videos stored on low-cost remote cameras.

Ethical considerations In this study: all visual data used is from the public domain; no information traceable to human individuals is collected or analyzed.

2 Motivations

2.1 Cold videos are already pervasive

Case study: Cold videos in real-world deployment We conduct an IRB-approved study examining existing camera deployment on PKU campus. Spanning 1 mi², the campus hosts tens of thousands of employees and operates more than 1,000 cameras. All captured videos are stored for a few months for retrospective queries before deletion. The camera deployment supports AI-based queries, e.g., object detection, *not* traceable to unique persons, and reviews by human analysts. We analyzed system logs spanning six continuous months: in over 3,000,000 hours of videos (5.4 PB) have been captured, only <0.005% video data from <2% cameras are queried.

Why are most videos cold? (1) Interesting video events are both unpredictable (thus the need for capturing excessive videos) and sparse (thus low chances for footage being queried). For example, severe traffic breakdown contributes to less than 5% of the time per day [89]; Foreign intelligence surveillance court only reviewed a tiny fraction of video for terrorism events [93]. (2) Analyzing videos is expensive: it still requires a GPU of a few thousand dollars for high-accuracy object detection over a video stream [59]. (3) In years to come, cheap cameras will produce more videos.

2.2 Target queries and their execution

We target ad-hoc queries [51, 59, 96, 100]. The query parameters, including object classes, video timespans, and expected accuracies, are specified at query time rather than video capture time. Such queries are known for flexibility.

High-accuracy object detection is essential Object detection is the core of ad-hoc queries [58]. Minor accuracy loss in object detection may result in substantial loss in query performance, as we will demonstrate in §8. While NNs significantly advance object detection, new models with higher accuracy demand much more compute. For instance, compared to YOLOv3 (2018) [77], CornerNet (2019) [64] improves Average Precision by 28% while being 5× more expensive.

Low-cost cameras cannot answer queries without cloud Cameras in real-world deployment are reported to be resource-constrained [65]. Low-cost cameras (<\$40) have wimpy cores, e.g., Cortex-A9 cores for YI Home Camera [18] and MIPS32 cores for WyzeCam [17]; their DRAM is no more than a few GBs [15, 16]. In recent benchmarks, they run state-of-the-art object detection at 0.1 FPS [8, 66], incapable of keeping up with video capture at 1–30 FPS [51, 59]. NN accelerators still cannot run high-accuracy object detection fast enough at low enough monetary cost, e.g., Intel’s Movidius (\$70) runs YOLOv3 at no faster than 0.5 FPS. In the foreseeable future, we expect that the resource gap between high-accuracy object detection and low-cost camera continues to exist.

2.3 A case for zero streaming

Streaming cold videos wastes bandwidth As discussed in §1, cameras are cheap while wireless spectrum is precious. Deploying streaming cameras on a shared network incurs poor experience [3, 11] and draws researcher attention [40, 100]. Dedicated networks are costly [14] and thus only suit a small number of cameras in critical locations. While wireless bandwidth grows, consumer demand grows even faster, e.g., 20× for VR/AR and 10× for gaming [5]. Cold video traffic should not contend with consumers for network bandwidth.

Streaming optimizations cannot offset the waste One may reduce FPS or resolution of streamed videos. Even if users tolerate the resultant lower query accuracy, the saved bandwidth is incomparable to the waste on overwhelmingly streamed cold videos, as we will experimentally show (§8). On-camera “early filters” [40, 42, 65] are still suboptimal when querying massive *cold* videos. (1) Without knowing query objects/parameters at video capture time, a camera may run a generic filter, e.g., discarding no-motion frames; it still streams substantial survival frames (e.g., consider a street-view camera). As stated above, most of these frames will remain cold and hence wasted. (2) The camera may run a large set of specific filters covering all possible query objects/parameters. Even if possible, this incurs a much higher compute cost to camera.

Edge processing does not justify streaming Cameras may stream to edge servers. Yet, streaming hundreds if not thousands of *always-on, cold* video streams, even if possible on certain wireless infrastructures, still wastes precious wireless spectrum at the edge [69]. Furthermore, deploying and managing video edge servers can be challenging and costly in

many scenarios, such as construction sites and remote farms.

Size	Yr.2017	Yr.2020	720p@30FPS	720p@1FPS
128GB	\$45	\$17	~11 days	~3 weeks
256GB	\$150	\$28	~3 weeks	~6 weeks

Table 1: Cheap μ SD cards on cameras retain long videos for humans to review [4] or for machines to analyze [51].

Cameras can retain videos long enough Table 1 shows the price of μ SD cards has been dropped by $2.6\times\text{--}5.4\times$ in the past few years. Cameras can retain videos for several weeks and for several months soon. Such a retention period is already adequate for most retrospective query scenarios, where videos are retained from a few weeks to a few months based on best practice and legal regulations [1, 2, 7, 10]. For privacy, many regulations *prohibit* video retention longer than a few months and mandate deletion afterwards [1, 7].

Our model & design scope To harness cold videos, we advocate for zero streaming. We focus on cold videos being queried for the first time and querying individual cameras. We intend our design to form the basis of future enhancement and extension, e.g., resource scheduling for multiple queries/users/tenants [40], caching for repetitive queries [95], exploiting past queries for refinement [41], and exploiting cross-camera topology [54]. We address limited compute resource on cameras [15] and limited network bandwidth [47]. We do not consider the cloud as a limiting factor, assuming it runs fast enough to process frames uploaded from cameras.

3 The DIVA Overview

Query types Concerning a specific camera, an ad-hoc query $(\mathcal{T}, \mathcal{C})$ covers a video timespan \mathcal{T} , typically hours or days, and an object class \mathcal{C} as detectable by modern NNs, e.g., any of the 80 classes of YOLOv3 [77]. As summarized in Table 2, DIVA supports three query types: **Retrieval**, e.g., “retrieve all images that contain buses from yesterday”; **Tagging**, e.g., “return all time ranges when any deer shows up in the past week”, in which the time ranges are returned as metadata but not images; **Counting**, e.g., “return the maximum number of cars that ever appear in any frame today”.

System components DIVA spans a camera and the cloud. Between them, the network connection is only provisioned at query time. To execute a query, a camera runs lightweight NNs, or operators, to *filter* or *rank* the queried frames for upload. On the uploaded frames, the cloud runs generic, high-accuracy object detection and materializes query results. Table 2 summarizes executions for different queries:

- The camera executes **rankers** for *Retrieval* and *max Count* queries. A ranker scores frames; a higher score suggests that a frame is more likely to contain *any* object of interest (for Retrieval) or *a large count* of such objects (for max Count).

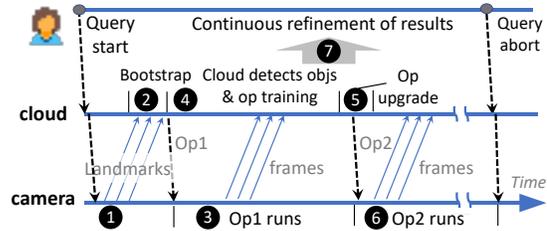


Figure 3: The workflow of a query’s execution.

- The camera executes **filters** for *Tagging* queries. A filter scores frames; it resolves any frame scored below/above two pre-defined thresholds as negative/positive, and deems other frames as unresolved. For each resolved frame, the camera uploads a positive/negative tag; the camera either uploads unresolved frames for the cloud to decide or defer them to more accurate filters on camera in subsequent passes.

Query execution Upon receiving a query, the cloud retrieves all landmarks in queried video as low-resolution thumbnails, e.g., 100×100 , with object labels and bounding boxes (Figure 3 1). The cloud uses landmarks: (1) *to estimate object spatial distribution*, e.g., “90% queried objects appear in a 100×100 region on the top-right”, which is crucial to query optimization (§4); (2) *as the initial training samples* for bootstrapping a family of camera operators (2). The camera filters/ranks frames and uploads the ranked or surviving frames (3). The cloud processes the uploaded frames and emits results, e.g., positive frames. It trains operators for higher accuracy (4). Observing resource conditions and positive ratios in uploaded frames, the cloud upgrades the operator on camera (5). With the upgraded operator, the camera continues to process remaining frames (6). Step (4)–(6) repeat until query abort or completion. Throughout the query, the cloud keeps refining the results presented to the user (7).

Notable designs (1) The camera processes frames in multiple passes, one operator in each pass. (2) The camera processes and uploads frames asynchronously. For instance, when the camera finishes ranking 100 out of total 1,000 frames, it may have uploaded the top 50 of the 100 ranked frames. This is opposed to common ranking which holds off frame upload until all the frames are ranked [38, 53, 61]. (3) The processing/upload asynchrony facilitates video exploration: it amortizes query delay over many installments of results; it pipelines query execution with user thinking [45]. Table 2 summarizes a user’s view of query results and the performance metrics. While such online query processing has been known [43, 71], we are the first applying it to visual data.

Limitations DIVA is not designed for several cases and may underperform: querying very short video ranges, e.g., minutes, for which simply uploading all queried frames may suffice without operators; querying non-stationary cameras for which landmarks may not yield accurate object distribution. DIVA is vulnerable to loss of video data in case of camera stor-

Type & Semantics	Execution	User's view of query results	Performance Metrics
Retrieval. Get positive video frames (i.e., containing C) within T	Camera: multipass ranking of frames Uploaded: ranked frames Cloud: object detection for identifying true positives	<ul style="list-style-type: none"> Positive frames being uploaded; Estimated % of positives retrieved 	The rate of the user receiving positive frames
Tagging. Get time ranges from T that contain C	Camera: multipass filtering of frames Uploaded: unresolved frames; tags of resolved frames Cloud: object detection to tag unresolved frames	<ul style="list-style-type: none"> A video timeline with pos/neg ranges; Tagging resolution, i.e., 1 in every N adjacent frames tagged 	The refining rate of tagging resolution seen by the user
Counting. Get max/mean/median count of C across all frames in T	Camera: multipass ranking (max) or random sampling (mean/median) of frames Uploaded: ranked or sampled frames Cloud: object detection to count objects	<ul style="list-style-type: none"> Running counts that converge to ground truth; % of frames processed; Estimated time to complete the query 	The rate of running counts converging to ground truth

Table 2: A summary of supported queries. \mathcal{T} is the queried video timespan; C is the queried object class



Figure 4: **Class spatial skews in videos.** In (a) Banff: 80% and 100% of cars appear in regions that are only 19% and 57% of the whole frame, respectively.



Figure 5: **Class spatial distribution can be estimated from sparse frames sampled over long video footage.** Among the three heatmaps: while sparse sampling over short footage (left) significantly differs from dense sampling of long footage (right), sparse sampling of long footage (middle) is almost equivalent to the right. Video: Tucson (see Table 4).

age failure. Users can mitigate such a risk via cross-camera data backup (RAID-like techniques) on the same local area network or by increasing camera deployment density.

4 Landmark Design

Surveillance cameras have a unique opportunity: to learn *object class distribution* from weeks of videos. We focus on **spatial skews**: objects of a given class are likely to concentrate on certain small regions on video frames. In examples of Figure 4(a)-(b), most cars appear near a stop sign; most persons appear in a shop's aisle. Such long-term skews are rarely tapped in prior computer vision work, which mostly focused on minute-long videos [52, 54, 78, 81, 102]. Compared to recent work that improved classifier performance by cropping video frames [40], DIVA takes a step further by automatically learning spatial skews from sparse frames with resource efficiency.

The design is backed up by three key observations. (1) One object class may exhibit different skews in different videos (Figure 4(a)-(c)); different classes may exhibit different skews

in the same video (Figure 4(c)). (2) The skews are pervasive: surveillance cameras cover long time spans and a wide field of view, where objects are small; in the view, objects are subject to social constraints, e.g., buses stop at traffic lights, or physical constraints, e.g., humans appear on the floor. (3) The skews can be learned through *sparse* frame samples, as exemplified by Figure 5.

To exploit such an opportunity, DIVA makes the following design choices. **High-accuracy object detection:** at capture time, the camera runs an object detector with the highest accuracy as allowed by the camera's hardware, mostly memory capacity. This is because camera operators crucially depend on the correctness of landmarks, i.e., the object labels and bounding boxes. We will validate this experimentally (§8.3). **Sparse sampling at regular intervals:** to accommodate slow object detection on cameras, the camera creates landmarks at long intervals, e.g., 1 in every 30 seconds in our prototype (§8). Sparse sampling is proven valid for estimating statistics of low-frequency signals [37], e.g., object occurrence in videos in our case. We will validate this (§8.3); without assuming a priori of object distribution, regular sampling ensures unbiased estimation of the distribution [79]. Given a priori, cameras may sample at corresponding random intervals for unbiased estimation.

Key idea: exploiting spatial skews for performance The cloud learns the object class distribution from landmarks of the queried video timespan. It generates a heatmap for spatial distribution (Figure 4). Based on the heatmap, the cloud produces camera operators consuming frame regions of different locations and sizes. Take Figure 4(a) as an example: a filter may consume bottom halves of all frames and accordingly filter frames with no cars; for Figure 4(b), a ranker may consume a smaller bounding box where 80% persons appear and rank frames based on their likelihood of containing more persons. Figure 6 shows that, by zooming into smaller regions, operators run faster and deliver higher accuracy. By varying input region locations/sizes, DIVA produces a set of operators with diverse costs/accuracies. By controlling the execution order of operators, DIVA processes "popular" frame regions prior to "unpopular" regions. DIVA never omits any region when it executes a query to completion to guarantee correctness.

What happens to instances uncaptured by landmarks?

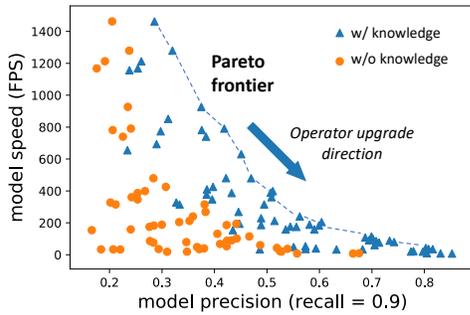


Figure 6: **On-camera operators benefit from long-term video knowledge substantially.** Each marker: an operator. For querying buses on video Banff (see Table 4).

Sparse by design, landmarks are not meant to capture all object instances; instead, they are used as inexact estimators and initial training samples. Reducing landmarks will degrade query speed, as we will experimentally quantify in §8.3. Doing so, however, does not affect query correctness or accuracy: the instances uncaptured by landmarks will be eventually processed by DIVA as a query goes on.

5 Online Operator Upgrade

5.1 The rationale

Three factors determine a query’s execution speed:

1. *Pending workloads*: the difficulty of the frames to be processed, i.e., how likely will the frame be mis-filtered or mis-ranked on camera.
2. *Camera operators*: cheap operators spend less time on each frame but are more likely to mis-filter/mis-rank frames, especially difficult frames. This is shown in Figure 6.
3. *Network condition*: the available uplink bandwidth.

The three factors interplay as follows.

- **Queries executed with on-camera rankers** A camera ranks and uploads frames asynchronously (§3). The key is to maximize the rate of true positive frames arriving at the cloud, for which the system must balance ranking speed/accuracy with upload bandwidth. (1) When the camera runs a *cheaper* ranker, it ranks frames at a much higher rate than uploading the frames; as a result, the cloud receives frames *hastily* selected from a *wide* video timespan. (2) When the camera runs an *expensive* ranker, the cloud receives frames selected *deliberately* from a *narrow* timespan. (3) The camera should never run rankers slower than upload, which is as bad as uploading unranked frames.

As an example, E_{CHEAP} and E_{EXP} on the top of Figure 7 compare two possible executions of the same query, running cheap/expensive rankers respectively. Shortly after the query starts (1), E_{CHEAP} swiftly explores more frames on camera; it outperforms E_{EXP} by discovering and returning more true

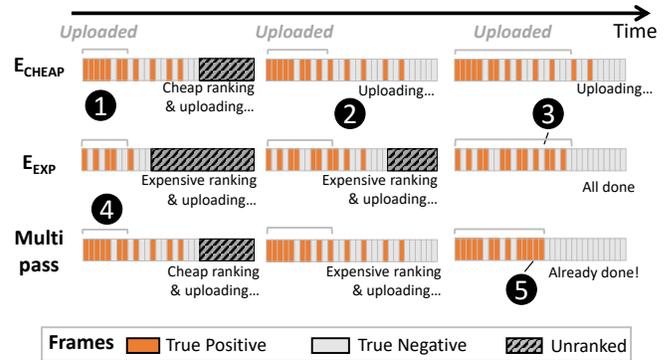


Figure 7: **Three alternative executions of a Retrieval query, showing multipass ranking (bottom) outperforms running individual rankers alone (top two).** Each row: snapshots of the upload queue at three different moments. In a queue: ranking/uploading frames from left to right.



Figure 8: **Cheap/expensive camera operators excel at different query stages.** Each subfigure: two alternative executions of the same query, showing query progress (bars) and the corresponding operator’s progress (arrows).

positive frames. As both executions proceed to harder frames (2), E_{CHEAP} makes more mistakes in ranking; it uploads an increasingly large ratio of negatives which wastes the execution time. By contrast, E_{EXP} ranks frames slower yet with much fewer mistakes, hence uploading fewer negatives. It eventually returns all positives earlier than E_{CHEAP} (3).

The microbenchmark in Figure 8(a) offers quantitative evidence. E1 spends *less* time ($0.7\times$) in returning the first 90% positives, but *more* time ($1.7\times$) in returning 99% positives. Furthermore, *lower* upload bandwidth favors a more *expensive* ranker, as the uploaded frames would contain a *higher* ratio of positives, better utilizing the precious bandwidth.

- **Queries executed with on-camera filters** The key is to maximize the rate of resolving frames on camera. Cheap filters excel on easy frames, resolving these frames fast with confidence. They are incapable on difficult frames, wasting time on attempting frames without much success in resolving. They would underperform *expensive* filters that spend more time per frame yet being able to resolve more frames.

The benchmark in Figure 8(b) shows two executions with cheap/expensive filters. Early in the query, E1 makes faster progress as the camera quickly resolves 50% of the frames ($4\times$ less time than E2). Later in the execution, E1 lags behind as the camera cannot resolve the remaining frames and must

upload them. By contrast, E2 resolves 82% of frames on camera and only uploads the remaining 18%. As a result, E2 takes $1.3\times$ less time in completing 90% and 99% of the query.

Summary & implications It is crucial for DIVA to pick operators with optimal cost/accuracy at query time. The choice not only varies across queries but also varies throughout a query’s execution: easy frames are processed early, leaving increasingly difficult frames that call for more expensive operators. DIVA should monitor pending frame difficulty and network bandwidth and upgrade operators accordingly.

5.2 Multipass, multi-operator execution

DIVA manages operators with the following techniques. (1) A camera processes frames iteratively with multiple operators. (2) The cloud progressively updates operators on camera, from cheaper ones to more expensive ones, as the direction shown in Figure 6. In picking operators, the cloud dynamically adapts operator speed to frame upload speed. (3) The cloud uses frames received in early execution stages to train operators for later stages; as the latter operators are more expensive, they require more training samples.

- **Multipass ranking** This is exemplified by the bottom execution in Figure 7. The camera first runs a cheap ranker, moving positives towards the front of the upload queue (4). Subsequently, the camera runs an expensive ranker, continuously reordering unsent frames in a smaller scope (5). Throughout the query, the camera first quickly uploads easy frames that are quickly ranked and slows down to vet difficult frames with expensive/accurate ranking. Notably, the cheaper ranker roughly prioritizes the frames as input for the expensive ranker, ensuring the efficacy of the expensive ranker. In actual query executions, a camera switches among 4–8 operators (§8).

- **Multipass filtering** The camera sifts undecided, unsent frames in multiple passes, each with a more expensive filter over a sample of the remaining frames. Throughout one query, early, cheaper filters quickly filter easier frames, leaving more difficult frames for subsequent filters to resolve.

6 Query Execution Planning

DIVA plans a concrete query execution by (1) the camera’s policy for selecting frames to process; (2) the cloud’s policy for upgrading on-camera operators. We now discuss them.

6.1 Executing Retrieval queries

Policy for selecting frames To execute the initial operator, the camera prioritizes fixed-length video spans (e.g., 1 hour) likely rich in positive frames, estimated based on landmark frames. In executing subsequent operators, the camera processes frames in their existing ranking as decided by earlier operators, as described in §5. The camera gives opportunities

to frames never ranked by prior operators, interleaving their processing with ranked frames with mediocre scores (0.5).

Policy for operator upgrade As discussed in §3, DIVA switches from cheap operators to expensive ones, and matches operator speed to frame upload rate. To capture an operator op ’s relative speed to upload, it uses one simple metric: the ratio between the two speeds, i.e., $f_{op} = FPS_{op}/FPS_{net}$. Operators with higher f_{op} tend to rapidly explore frames while others tend to exploit slowly. The operator speed FPS_{op} is profiled offline. (1) **Selecting the initial operator** In general, DIVA should fully utilize the upload bandwidth with positive frames. As positive frames are scattered in the queried video initially, the camera should explore all frames sufficiently fast. Otherwise, it would either starve the uplink or knowingly upload negative frames. Based on this idea, the cloud picks the most accurate operator from the ones that are fast enough, i.e., $f_{op} \times R_{pos} > 1$, where R_{pos} is the ratio of positives in the queried video, estimated from landmarks. (2) **When to upgrade: current operator losing its vigor** The cloud upgrades operators either when the current operator finishes processing all frames, or the cloud observes a continuous quality decline in recently uploaded frames, an indicator of the current operator’s incapability. To decide the latter, DIVA employs a rule: the positive ratio in recently uploaded frames are $k\times$ (default: 5) lower than the frames uploaded at the beginning; (3) **Selecting the next operator: slow down exponentially** Since the initial operator promotes many positives towards the front of the upload queue, subsequent operators, scanning from the queue front, likely operate on a larger fraction of positives. Accordingly, the cloud picks the most accurate operator among much slower ones, s.t. $f_{op(i+1)} > \alpha \times f_{op(i)}$, where α controls speed decay in subsequent operators. A larger α leads to more aggressive upgrade: losing more speed for higher accuracy. In the current prototype, we empirically choose $\alpha = 0.5$. Since f is relative to FPS_{net} measured at every upgrade, the upgrade adapts to network bandwidth change during a query.

6.2 Executing Tagging queries

Recall that for *Tagging*, a camera runs multipass filtering; the objective of each pass is to tag, as positive (P) or negative (N), at least one frame from every K adjacent frames. We call K the group size; DIVA pre-defines a sequence of group sizes as refinement levels, e.g., $K = 30, 10, \dots, 1$. As in prior work [51, 58, 59], the user specifies tolerable error as part of her query, e.g., 1% false negative and 1% false positive; DIVA trains filters with thresholds to meet the accuracy.

Policy for selecting frames The goal is to quickly tag easy frames in individual groups while balancing the workloads of on-camera processing and frame upload. An operator op works in two stages of each pass. i) *Rapid attempting*. op scans all the groups; it attempts one frame per group; if it succeeds, it moves to the next group; it adds undecidable

frames (U) to the upload queue. ii) *Work stealing*. op steals work from the end of upload queue. For an undecidable frame f belonging to a group g , op attempts other untagged frames in g ; once it succeeds, it removes f from the upload queue as f no longer needs tagging in the current pass. After one pass, the camera switches to the next refinement level (e.g., $10 \rightarrow 5$). It keeps all the tagging results (P, N, U) while cancels all pending uploads. It re-runs the frame scheduling algorithm until it meets the finest refinement level or query terminated.

Policy for operator upgrade Given an operator op and γ_{op} , the ratio of frames it can successfully tag, DIVA measures op 's efficiency by its effective tagging rate, $FPS_{op} \times \gamma_{op} + FPS_{net}$, as a sum of op 's successful tagging rate and the uploading rate. As part of operator training, the cloud estimates γ_{op} for all the candidate operators by testing them on all landmarks (early in query) and uploaded frames (later in query). To select every operator, initial or subsequent, the cloud picks the candidate with the highest effective tagging rate. The cloud upgrades operators either when the current operator has attempted all remaining frames or another candidate having an effective tagging rate $\beta \times$ or higher (default value 2).

6.3 Executing Counting queries

Max Count: Policy for selecting frames To execute the initial operator, the camera randomly selects frames to process, avoiding the worst cases that the max resides at the end of the query range. For subsequent operators, the camera processes frames in existing ranking decided by earlier operators.

Max Count: Policy for operator upgrade As the camera runs rankers, the policy is similar to that for *Retrieval* with a subtle yet essential difference. To determine whether the current operator shall be replaced, the cloud must assess the quality of recently uploaded frames. While for *Retrieval*, DIVA conveniently measures the quality as the ratio of *positive* frames, the metric does not apply to *max Count*, which seeks to discover *higher* scored frames. Hence, DIVA adopts the Manhattan distance as a quality metric among the permutations from the ranking of the uploaded frames (as produced by the on-camera operator) and the ranking that is re-computed by the cloud object detector. A higher metric indicates worse quality hence more urgency for the upgrade.

Average/Median Count: no on-camera operators After the initial upload of landmarks, the camera randomly samples frames in queried videos and uploads them for the cloud to refine the average/median statistics. To avoid any sampling bias, the camera does not prioritize frames; it instead relies on the Law of Large Numbers (LLN) [48] to approach the average/median ground truth through continuous sampling.

Cameras	Rpi3 (default): Raspberry Pi 3 (\$35). 4xCortex-A53, 1GB DRAM Odroid: XU4 (\$49) 4xCortexA15 & 4xCortexA7, 2GB DRAM		
CloudServer	2x Intel Xeon E5-2640v4, 128GB DRAM GPU: Nvidia Titan V		
(a) <i>Hardware platforms</i>			
	Cam:Landmarks	Cam:Query	Cloud:Query
ClondOnly	–	Only upload frames	Yv3 on all uploaded frames
OptOp	Yv3 every 30 secs	Run one optimal op	
PreIndexAll	YTiny every sec	Parse YTiny result	
DIVA	Yv3 every 30 secs	Multi passes & ops	

(b) *DIVA and the baselines*. The table summarizes their executions for capture and query. NNs: Yv3 – YOLOv3, high accuracy (mAP=57.9); YTiny – YOLOv3-tiny, low accuracy (mAP=33.1).

Table 3: Experiment configurations

	Name	Object	Description
T	JacksonH [25]	car	A busy intersection in Jackson Hole, WY
	JacksonT [26]	car	A night street in Jackson Hole, WY
	Banff [20]	bus	A cross-road in Banff, Alberta, Canada
	Mierlo [29]	truck	A rail crossing in Netherlands
	Miami [28]	car	A cross-road in Miami Beach, FL
	Ashland [19]	train	A level crossing in Ashland, VA
O	Shibuya [31]	bus	An intersection in Shibuya (渋谷), Japan
	Chaweng [22]	bicycle	Absolut Ice Bar (outside) in Thailand
	Lausanne [27]	car	A pedestrian plaza in Lausanne, Switzerland
	Venice [32]	person	A waterfront walkway in Venice, Italy
I	Oxford [30]	bus	A street beside Oxford Martin school, UK
	Whitebay [33]	person	A beach in Virgin Islands
	CoralReef [23]	person	An aquarium video from CA
W	BoatHouse [21]	person	A retail store from Jackson Hole, WY
	Eagle [24]	eagle	A tree with an eagle nest in FL

Table 4: 15 videos used for test. Each video: 720P at 1FPS lasting 48 hours. Column 1: video type. T – traffic; O/I – outdoor/indoor surveillance; W – wildlife.

7 Implementation and Methodology

Operators We architect on-camera operators as variants of AlexNet [63]. We vary the number of convolutional layers (2–5), convolution kernel sizes (8/16/32), the last dense layer's size (16/32/64); and the input image size ($25 \times 25 / 50 \times 50 / 100 \times 100$). We empirically select 40 operators to be trained by DIVA online; we have attempted more but see diminishing returns. These operators require small training samples (e.g., 10K images) and run fast on camera.

Background subtraction filters static frames at low overhead [51]. DIVA employs a standard technique [12]: during video capture, a camera detects frames that have little motion ($< 1\%$ foreground mask) and omits them in query execution. On our camera hardware (Table 3), background subtraction is affordable in real time during capture. For fair comparisons, we augment all baselines with background subtraction.

Videos & Queries We test DIVA on 15 videos captured from 15 live camera feeds (Table 4). Each video lasts continuous 48 hours including daytime and nighttime, collected between Oct. 2018 to Mar. 2019. We preprocess all videos to be 720P at 1 FPS, consistent with prior work [51]. We test *Retrieval/Tagging/Counting* queries on 6/6/3 videos. We intentionally choose videos with disparate characteristics and hence different degrees of difficulty. For instance, Whitebay

is captured from a close-up camera, containing clear and large persons; Venice is captured from a high camera view and hence contains blurry and small persons. For each video, we pick a representative object class to query; across videos, these classes are diverse. For *Tagging*, we set query error to be $< 1\%$ FN/FP as prior work did [59].

A query's accuracy is reflected by its execution progress. For retrieval/counting, we report accuracy as the fraction of positive frames returned. There is no false positive because the cloud always runs the high-accuracy object detector as a "safety net", of which the output is regarded as the ground truth. For tagging, we report accuracy as query errors, meaning the percentage of frames mistakenly tagged. To issue a query, the user sets the target error, which by default is 1% as in prior work. Table 2 and §6 provide more details.

Test platform & parameters As summarized in Table 3(a), we test on embedded hardware similar to low-cost cameras [15, 16]. We use Rpi3 as the default camera hardware and report its measurement unless stated otherwise. During query execution, both devices set up a network connection with 1MB/s default bandwidth to emulate typical WiFi condition [47]. Note that this network bandwidth is *not* meant for streaming; it is only for a camera while the camera is being queried. We run YOLOv3 as the high-accuracy object detector on camera and cloud (Table 3(b)). In calculating accuracy, we use the output of YOLOv3 as the ground truth as in prior work [51, 59]. On Rpi3, we partition YOLOv3 into three stages, each fitting into DRAM separately. We will study alternative models, landmarks, and resources in §8.3.

Baselines As summarized in Table 3(b), we compare DIVA with three alternative designs augmented with background subtraction and only process/transmit non-static video frames.

- **CloudOnly**: a naive design that uploads all queried frames at query time for the cloud to process.
- **OptOp**: in the spirit of NoScope [59], the camera runs only one ranker/filter specialized for a given query, selected by a cost model for minimizing full-query delay. To make OptOp competitive, we augment it with landmark frames to reduce the operator training cost. Compared to DIVA, OptOp's key differences are the lack of operator upgrade and the lack of operator optimization by long-term video knowledge.
- **PreIndexAll**: in the spirit of Focus [51], the camera runs a cheap yet generic object detector on all frames. We pick YOLOv3-tiny (much cheaper than YOLOv3) as the detector affordable by Rpi3 in real time (1 FPS). The detector plays the same role as an operator in DIVA, except that it runs at capture time: for *Retrieval* and *Counting*, the detector's output scores are used to prioritize frames to upload at query time; for *Tagging*, the output is used to filter the frames that have enough confidence. PreIndexAll implements all run-time features of Focus except feature clustering. We left out clustering because we find it performs poorly on counting queries. Compared to DIVA, PreIndexAll's key differences

are: it answers queries solely based on the indexes built at capture time; it requires no operator training or processing actual images at query time.

8 Evaluation

8.1 End-to-end performance

Full query delay is measured as: (*Retrieval*) the time to receive 99% positive frames as in prior work [51]; (*Tagging*) the time taken to tag every frame; (*Counting*) the time to reach the ground truth (max) or converge within 1% error of the ground truth (avg/median). Overall, DIVA delivers good performance and outperforms the baselines significantly.

- **Retrieval** (Figure 9(a)). On videos each lasting 48 hours, DIVA spends $\sim 1,900$ seconds on average, i.e., $89\times$ of video realtime. On average, DIVA's delay is $3.8\times$, $3.1\times$, and $2.0\times$ shorter than CloudOnly, PreIndexAll, and OptOp, respectively.
- **Tagging** (Figure 9(b)). DIVA spends ~ 581 seconds on average ($297\times$ realtime). This delay is $16.0\times$, $2.1\times$, and $4.3\times$ shorter than CloudOnly, PreIndexAll, and OptOp, respectively.
- **Counting** (Figure 10). DIVA's average/median take several seconds to converge. For *average Count*, DIVA's delay is $65.1\times$ and up to three orders of magnitude shorter than CloudOnly and PreIndexAll. For *median Count*, DIVA's delay is $68.3\times$ shorter than the others. For *max Count*, DIVA spends 34 seconds on average ($635\times$ realtime), which is $5.8\times$, $5.0\times$, and $1.3\times$ shorter than CloudOnly, PreIndexAll, and OptOp.

Query progress DIVA makes much faster progress in most time of query execution. It *always* outperforms CloudOnly and OptOp during *Retrieval/Tagging* (Figure 9). It *always* outperforms alternatives in median/average count (Figure 10).

Why DIVA outperforms the alternatives? The alternatives suffer from the following. (1) Inaccurate indexes. PreIndexAll resorts to inaccurate indexes (YOLOv3-tiny) built at capture time. Misled by them, *Retrieval* and *Tagging* upload too much garbage; *Counting* includes significant errors in the initial estimation, slowing down convergence. (2) Lack of long-term knowledge. OptOp's operators are either slower or less accurate than DIVA, as illustrated in Figure 6. (3) One operator does not fit an entire query. Optimal at some point (e.g., 99% Retrieval), the operator runs too slow on easy frames which could have been done by cheaper operators.

Why DIVA underperforms (occasionally)? On short occasions, DIVA may underperform PreIndexAll at early query stages, e.g., BoatHouse in Figure 9. Reasons: (1) PreIndexAll's inaccurate indexes may be correct on *easy* frames; (2) PreIndexAll does not pay for operator bootstrapping as DIVA. Nevertheless, PreIndexAll's advantage is transient. As easy frames are exhausted, indexes make more mistakes on the remaining frames and hence slow down the query.

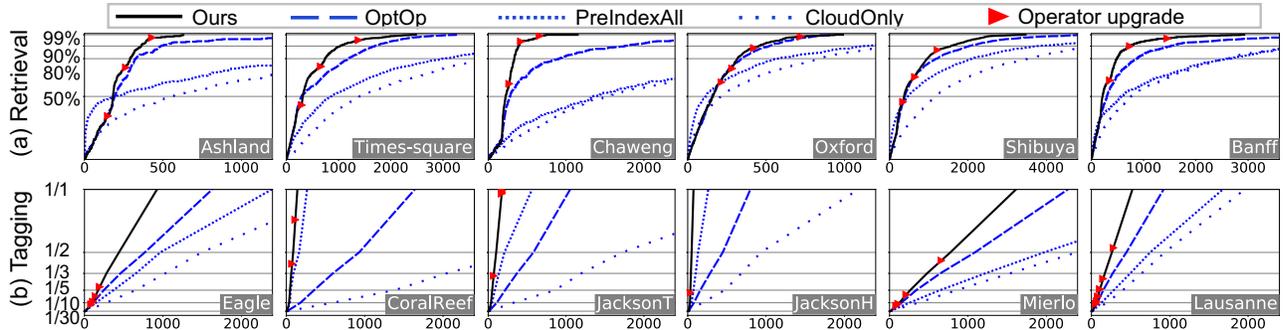


Figure 9: **On Retrieval and Tagging queries, DIVA shows good performance and outperforms the alternatives.** x-axis for all: query delay (secs). y-axis for (a): % of retrieved instances; for (b): refinement level (1/N frames).

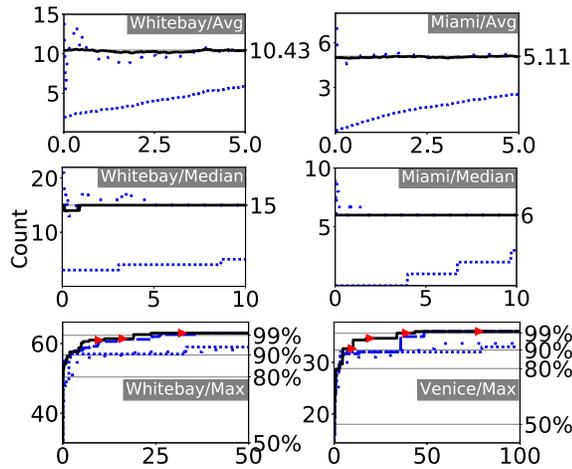


Figure 10: **On Counting queries, DIVA shows good performance and outperforms the alternatives.** Legend: see Figure 9. x-axis for all: query delay (secs). y-axis for left plots: count; for top two right plots: ground truth for avg/median queries; for bottom right plot: % of max value.

Can DIVA outperform under different network bandwidths at query time? Table 5 summarizes DIVA’s query delays at 9 bandwidths evenly spaced in [0.1 MB/s, 10 MB/s] which cover typical WiFi bandwidths [9]. We have observed that: on lower bandwidths, DIVA’s advantages over baselines are more significant; at high bandwidths, DIVA’s advantages are still substantial ($>2\times$ in most cases) yet less pronounced. The limitation is not in DIVA’s design but rather its current NNs: we find it difficult to train operators that are both fast enough to keep up with higher upload bandwidth and accurate enough to increase the uploaded positive ratio proportionally.

vs. “all streaming”: query speed As “all streaming” uploads all videos to the cloud before a query starts, the query speed is bound by cloud GPUs but not network bandwidth. With our default experiment setting (1 GPU and 1MB/s network bandwidth), “all streaming” still runs queries much slower than zero streaming. Adding more cloud GPUs will eventually make “all streaming” run faster than DIVA.

	Retrieval	Tagging	Count/Max	Count/Avg&Med
CloudOnly	4.5/14.9/52.2	3.61/3.9/5.1	2.8/21.1/42.5	6.9/83.4/439.2
OptOp	2.2/4.1/4.9	2.0/2.3/2.6	1.2/1.5/2.1	6.9/83.4/439.2*
PreIndexAll	1.9/3.8/11.6	3.2/3.6/4.9	1.2/8.9/18.2	2.5/14.0/41.3

*: Fall back to CloudOnly as the camera does not execute NN for these query types

Table 5: DIVA’ performance (speedup) with various bandwidths. Numbers: min/median/max of times (\times) of query delay reduction compared to baselines (rows). Averaged on all videos and 9 bandwidths in 0.1MB/s–10MB/s.

vs. “all streaming”: network bandwidth saving Compared to streaming all videos (720P 1FPS) at capture time, DIVA saves traffic significantly, as shown in Figure 11. When only as few as 0.005% of video is queried as in our case study (§2), the saving

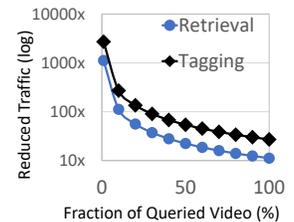


Figure 11: **DIVA significantly reduces network traffic compared to “all streaming”.** Results averaged over all videos.

is over three orders of magnitude. Even if all captured videos are queried, DIVA saves more than $10\times$, as its on-camera operators skip uploading many frames. Among the bandwidth reduction brought by DIVA, only less than 30% attributes to the background subtraction technique. It shows that the disadvantage of “all streaming” is fundamental: streaming optimizations may help save the bandwidth (upmost several times [96]) but cannot offset the waste, as discussed in §2.3.

Training & shipping operators For each query, DIVA trains ~ 40 operators, of which ~ 10 are on the Pareto frontier. The camera switches among 4–8 operators, which run at diverse speeds ($27\times$ – $1,000\times$ realtime) and accuracies. DIVA chooses very different operators for different queries. Training one operator typically takes 5–45 seconds on our test platform and requires 5k frames (for bootstrapping) to 15k frames (for stable accuracy). Operators’ sizes range from 0.2–15 MB. Sending an operator takes less than ten seconds. Only the delay in training and sending the first operator (≤ 40 seconds) adds to the query delay which is included in Figure 9/10.

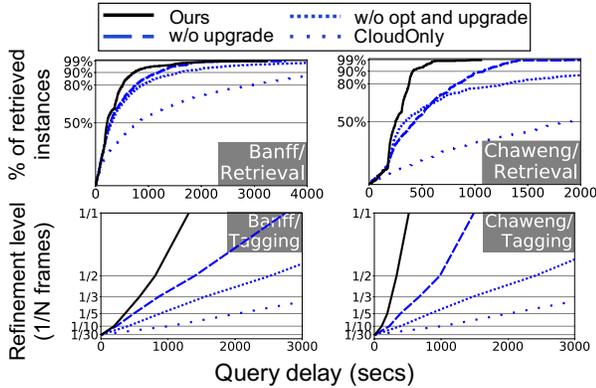


Figure 12: DIVA’s both key techniques – optimization with long-term video knowledge (opt) and operator upgrade (upgrade), contribute to performance significantly.

Subsequent operators are trained and transmitted in parallel to query execution. Their delays are hidden from users.

DIVA elasticity Due to DIVA’s design, the computing resources available on low-cost cameras are used efficiently at both capture and query time. Thanks to its elastic execution, it can avoid interference with a camera’s surveillance task, notably video encoding and storage. For instance, DIVA can produce denser/sparser landmarks per its CPU time allocated by the camera OS. According to our experiments on Raspberry Pi 3B+, recording video at 720P and 30 FPS only uses less than 2% of CPU time, which is negligible as compared to NN execution. We reserve a small fraction of CPU time to surveillance using *cgroup* and observe no frame drop in the surveillance task and negligible slowdown in NN execution.

8.2 Validation of query execution design

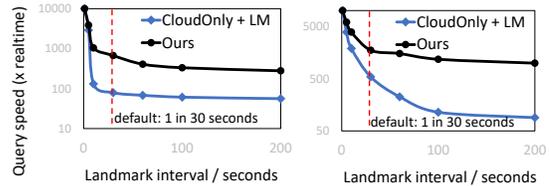
The experiments above show DIVA’s substantial advantage over *OptOp*, coming from a combination of two techniques – optimizing queries with long-term video knowledge (“*Long-term opt*”, §4) and operator upgrade (“*Upgrade*”, §5). We next break down the advantage by incrementally disabling the two techniques in DIVA. Figure 12 shows the results.

Both techniques contribute to significant performance. For instance, disabling *Upgrade* increases the delay of retrieving 90% instances by 2× and that of tagging 1/1 frames by 2×-3×. Further disabling *Long-term Opt* increases the delay of Retrieval by 1.3×-2.1× and that of tagging by 1.6×-3.1×. Both techniques disabled, DIVA still outperforms *CloudOnly* with its single non-optimized operator.

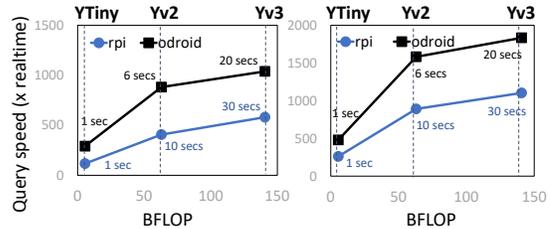
Upgrade’s benefit is universal; Long-term opt’s benefit is more dependent on queries, i.e., the skews of the queried object class in videos. For instance, DIVA’s benefit is more pronounced on Chaweng, where small bicycles only appear in a region in 1/8 size of the entire frame, than Ashland, where large trains take 4/5 of the frame. With stronger skews in Chaweng, DIVA trains operators that are more accurate



(a) DIVA’s performance degrades significantly with less accurate landmarks (produced by Yv2 and YTiny), which can be even worse than no landmarks at all (“w/o LM”).



(b) DIVA’s performance degrades slowly with sparser landmarks. The y-axis is logarithmic.



(c) On given camera hardware (Rpi3/Odroid), sparser yet more accurate LMs always improve DIVA’s performance. Landmark intervals annotated along curves.

Figure 13: **Validation of landmark design.** In (a)/(b)/(c): Left – *Retrieval* on Chaweng; Right – *Tagging* on JacksonH.

and run faster. This also accounts for DIVA’s varying (yet substantial) advantages over the alternatives (Figure 9).

8.3 Validation of landmark design

Next, we deviate from the default landmark parameters (Table 3) to validate the choice of sparse-but-sure landmarks.

DIVA hinges on accurate landmarks. As shown in Figure 13(a), modestly inaccurate landmarks (as produced by YOLOv2; 48.1 mAP) increase delays for Q1/Q2 by 45% and 17%. Even less accurate landmarks (by YOLOv3-tiny; 33.1 mAP) increase the delays significantly by 5.3× and 4.3×. Perhaps surprisingly, such inaccurate landmarks can be worse than no landmarks at all (“w/o LM” in Figure 13): when a query starts, a camera randomly uploads unlabeled frames for the cloud to bootstrap operators. *Why inaccurate landmarks hurt so much?* They (1) provide wrong training samples; (2) lead to incorrect observation of spatial skews which further mislead frame cropping; and (3) introduce large errors into initial statistics, making convergence harder.

DIVA tolerates longer landmark intervals. As shown in Figure 13(b), DIVA’s *Retrieval* and *Tagging* performance slowly degrade with longer intervals. Even with an infinite

interval, i.e., “w/o LM” in Figure 13(a), the slowdown is no more than $3\times$. On *Counting*, the performance degradation is more pronounced: $5\times$ longer intervals for around $15\times$ slow down. Yet, such degradation is still much smaller than one from inaccurate landmarks (two orders of magnitude). The reason is that, with longer LM intervals DIVA has to upload additional frames in full resolution ($\sim 10\times$ larger than LMs) when a query starts for bootstrapping operators; such a one-time cost, however, is amortized over the full query.

Create the most accurate landmarks possible Should a camera build denser yet less accurate landmarks or sparser yet more accurate ones? Figure 13(c) suggests the latter is always preferred, because of DIVA’s high sensitivity to landmark accuracy and low sensitivity to long landmark intervals.

DIVA on wimpy/brawny cameras DIVA suits wimpy cameras that can only generate sparse landmarks. Some cameras may have DRAM smaller than a high-accuracy NN (e.g., ~ 1 GB for YOLOv3); fortunately, recent orthogonal efforts reduce NN sizes [56]. Wimpier cameras will further disadvantage the alternatives, e.g., `PreIndexAll` will produce even less accurate indexes. On higher-end cameras (a few hundred dollars each [13]) that DIVA is *not* designed for, DIVA still shows benefits, albeit not as pronounced. High-end cameras can afford more computation at capture time. i) They may run `PreIndexAll` with improved index accuracy. In Figure 13(a), running YOLOv2 on all captured frames (`PreIndexAll+Yv2`), DIVA’s performance gain is $1.9\times$ (left) or even $0.6\times$ (right). ii) These cameras may generate denser landmarks and rely on the cloud for the remaining frames. Figure 13(b) shows, with one landmark every 5 seconds, DIVA’s advantage is $1.5\times$.

9 Related Work

Optimizing video analytics The CV community has studied video analytics for decades, e.g., for online training [83, 84] and active learning [57]. They mostly focus on improving analytics accuracy on short videos [44, 60, 68, 78, 81, 102] while missing opportunities in exploiting long-term knowledge (§4). These techniques alone cannot address the systems challenges we face, e.g., network limit or frame scheduling. A common theme of recent work is to trade accuracy for lower cost: VStore [96] does so for video storage; Pakha *et al.* [70] do so for network transport; Chameleon [55] and VideoStorm [52, 99] do so with video formats. DIVA’s operators as well exploit accuracy/cost tradeoffs. Multiple systems analyze archival videos on servers [58, 62, 73, 80, 96]. DIVA analyzes archival videos on *remote* cameras and embraces new techniques. ML model cascade is commonly used for processing a stream of frames [39, 59, 85]: in processing a frame, it keeps invoking a more expensive operator if the current operator has insufficient confidence. This technique, however, mismatches exploratory analytics, for which DIVA uses one operator to process many frames in one pass and

produces inexact yet useful results for all of them.

Edge video analytics To reduce cloud/edge traffic, computation is partitioned, e.g., between cloud/edge [40, 76, 97], edge/drone [91], and edge/camera [100]. Elf [94] executes counting queries completely on cameras. Most work targets live analytics, processes frames in a streaming fashion and trains NNs ahead of time. DIVA spreads computation between cloud/cameras but takes a disparate design point (zero streaming) that are inadequate in prior systems. CloudSeg [92] reduces network traffic by uploading low-resolution frames and recovering them via super resolution. DIVA eliminates network traffic at capture time at all.

Online Query Processing Dated back in the 90s, online query processing allows users to see early results and control query execution [49, 50]. It is proven effective in large data analytics, such as MapReduce [43]. DIVA retrofits the idea for video queries and accordingly contributes new techniques, e.g., operator upgrade, to support the online fashion. DIVA could borrow UI designs from existing online query engines.

WAN Analytics To query geo-distributed data, recent proposals range from query placement to data placement [74, 86–88, 90]. JetStream [75] adjusts data quality to meet network bandwidth; AWStream [98] facilitates apps to systematically trade-off analytics accuracy for network bandwidth. Like them, DIVA adapts to network; unlike them, DIVA does so by changing operator upgrade plan, a unique aspect in video analytics. DIVA targets resource-constrained cameras, which are unaddressed in WAN analytics.

10 Conclusions

Zero streaming shifts most compute from capture time to query time. We build DIVA, an analytics engine for querying cold videos on remote, low-cost cameras. At capture time, DIVA builds sparse but sure landmarks; at query time, it refines query results by continuously updating on-camera operators. Our evaluation of three types of queries shows that DIVA can run at more than $100\times$ video realtime under typical wireless network and camera hardware.

Acknowledgments

Mengwei Xu was supported by National Key R&D Program of China under grant number 2020YFB1805500, the Fundamental Research Funds for the Central Universities, and National Natural Science Foundation of China under grant numbers 62032003, 61922017, and 61921003. Tiantu Xu and Felix Xiaozhu Lin were supported in part by NSF awards #1846102 and #1919197. We thank our shepherd, Michael Kozuch, and the anonymous ATC reviewers for their useful suggestions.

References

- [1] The european data protection supervisor video-surveillance guidelines. https://edps.europa.eu/sites/edp/files/publication/10-03-17_video-surveillance_guidelines_en.pdf, 2010.
- [2] Tufts: Video security university policy. <https://publicsafety.tufts.edu/policies/video-security/>, 2014.
- [3] Wireless cameras slowing router too much. <https://community.netgear.com/t5/Nighthawk-WiFi-Routers/Wireless-cameras-slowing-router-too-much/td-p/513047>, 2015.
- [4] Understanding ip surveillance camera bandwidth. <https://www.fortinet.com/content/dam/fortinet/assets/white-papers/wp-ip-surveillance-camera.pdf>, 2017.
- [5] The zettabyte era: Trends and analysis. <https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html>, 2017.
- [6] International trends in video surveillance. <https://cms.uitp.org/wp/wp-content/uploads/2020/06/18-07Statistics-Brief-Videosurveillance-web.pdf>, 2018.
- [7] New case law on retention periods for video surveillance at the workplace. <https://www.twobirds.com/en/news/articles/2018/germany/new-case-law-on-retention-periods-for-video-surveillance-at-the-workplace>, 2018.
- [8] Running yolo detection on raspberry pi. <http://raspberrypi4u.blogspot.com/2018/10/raspberry-pi-yolo-real-time-object.html>, 2018.
- [9] The state of wifi vs mobile network experience as 5g arrives. https://www.opensignal.com/sites/opensignal-com/files/data/reports/global/data-2018-11/state_of_wifi_vs_mobile_opensignal_201811.pdf, 2018.
- [10] Video surveillance laws: Video retention requirements by state. <https://www.verkada.com/blog/surveillance-laws-video-retention-requirements/>, 2018.
- [11] Wifi cameras. <https://www.security-camera-warehouse.com/ip-camera/wifi-enabled/>, 2018.
- [12] Background subtraction. https://docs.opencv.org/3.4.0/db/d5c/tutorial_py_bg_subtraction.html, 2019.
- [13] Build intelligent ideas with our platform for local ai. <https://coral.withgoogle.com/>, 2019.
- [14] Comcast business internet data plan. <https://www.business.org/services/internet/comcast-business-internet-review/>, 2019.
- [15] Hisilicon ip camera specifications. <http://www.hisilicon.com/en/Products/ProductList/Surveillance>, 2019.
- [16] Wyze camera specifications. <https://www.wyze.com/wyze-cam/specs/>, 2019.
- [17] Wyze camera v2 1080p. <https://www.wyze.com/product/wyze-cam-v2/>, 2019.
- [18] Yi home camera. <https://www.amazon.com/YI-Security-Surveillance-Monitor-Android/dp/B01CW4AR9K>, 2019.
- [19] Youtube live streaming: Ashland. <https://www.youtube.com/watch?v=e47XhLmZhFk>, 2019.
- [20] Youtube live streaming: Banff. <https://youtu.be/9HwSNgcdQ7k>, 2019.
- [21] Youtube live streaming: Boathouse. <https://www.youtube.com/watch?v=TXw7CyY0TbU&t=0s>, 2019.
- [22] Youtube live streaming: Chaweng. https://www.youtube.com/watch?v=tihJ58_qiH0, 2019.
- [23] Youtube live streaming: Coralreef. <https://youtu.be/WY0e8SfQbac>, 2019.
- [24] Youtube live streaming: Eagle. https://www.youtube.com/watch?v=Q_OrM8o2k6I, 2019.
- [25] Youtube live streaming: Jackson hole. <https://youtu.be/2wnU2Kp7quQ>, 2019.
- [26] Youtube live streaming: Jackson town. <https://www.youtube.com/watch?v=1EiC9bvVGnk>, 2019.
- [27] Youtube live streaming: Lausanne. <https://www.youtube.com/watch?v=7uF7DsUQ9vc>, 2019.
- [28] Youtube live streaming: Miami. <https://www.youtube.com/watch?v=0dctq-YjAdc>, 2019.
- [29] Youtube live streaming: Mierlo. <https://www.youtube.com/watch?v=HbtBgxFkDHU>, 2019.
- [30] Youtube live streaming: Oxford. <https://www.youtube.com/watch?v=St7aTfoIdYQ>, 2019.

- [31] Youtube live streaming: Shibuya. <https://youtu.be/PmrWwYt1AVQ>, 2019.
- [32] Youtube live streaming: Venice. <https://www.youtube.com/watch?v=JqUREqYduHw>, 2019.
- [33] Youtube live streaming: Whitebay. <https://www.youtube.com/watch?v=LXWVYoBluT4>, 2019.
- [34] Zosi camera. <https://www.amazon.com/ZOSI-1280TVL-Security-Weatherproof-Surveillance/dp/B01DF6LJZK>, 2019.
- [35] Aloÿs Augustin, Jiazi Yi, Thomas Clausen, and William Townsley. A study of lora: Long range & low power networks for the internet of things. *Sensors*, 16(9):1466, 2016.
- [36] David Beymer, Philip McLauchlan, Benjamin Coifman, and Jitendra Malik. A real-time computer vision system for measuring traffic parameters. In *Proceedings of IEEE computer society conference on computer vision and pattern recognition (CVPR)*, pages 495–501, 1997.
- [37] T. Blu, P. Dragotti, M. Vetterli, P. Marziliano, and L. Coulot. Sparse sampling of signal innovations. *IEEE Signal Processing Magazine*, 25(2):31–40, March 2008.
- [38] Christian Böhm, Bernhard Braunmüller, Florian Krebs, and Hans-Peter Kriegel. Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. In *ACM SIGMOD Record*, volume 30, pages 379–388, 2001.
- [39] Zhaowei Cai, Mohammad Saberian, and Nuno Vasconcelos. Learning complexity-aware cascades for deep pedestrian detection. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 3361–3369, 2015.
- [40] Christopher Canel, Thomas Kim, Giulio Zhou, Conglong Li, Hyeontaek Lim, David G. Andersen, Michael Kaminsky, and Subramanya R. Dulloor. Scaling video analytics on constrained edge nodes. In *Proceedings of the 2nd SysML Conference (SysML)*, 2019.
- [41] Kaushik Chakrabarti, Kriengkrai Porkaew, and Sharad Mehrotra. Efficient query refinement in multimedia databases. In *ICDE Conference*, January 2000. Poster paper.
- [42] Tiffany Yu-Han Chen, Lenin S. Ravindranath, Shuo Deng, Paramvir Victor Bahl, and Hari Balakrishnan. Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices. In *13th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2015.
- [43] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 21–21, 2010.
- [44] Boyuan Feng, Kun Wan, Shu Yang, and Yufei Ding. SECS: efficient deep stream processing via class skew dichotomy. *CoRR*, abs/1809.06691, 2018.
- [45] Ziqiang Feng, Shilpa George, Jan Harkes, Padmanabhan Pillai, Roberta Klatzky, and Mahadev Satyanarayanan. Eureka: Edge-based discovery of training data for machine learning. *IEEE Internet Computing*, PP:1–1, 01 2019.
- [46] Ziqiang Feng, Junjue Wang, Jan Harkes, Padmanabhan Pillai, and Mahadev Satyanarayanan. Eva: An efficient system for exploratory video analysis. *SysML*, 2018.
- [47] Bo Han, Feng Qian, Lusheng Ji, and Vijay Gopalakrishnan. Mp-dash: Adaptive video streaming over preference-aware multipath. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, pages 129–143, New York, NY, USA, 2016. ACM.
- [48] Michiel Hazewinkel. *Encyclopaedia of Mathematics*. Springer Netherlands, 1988.
- [49] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas. Interactive data analysis: the control project. *Computer*, 32(8):51–59, Aug 1999.
- [50] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 171–182, New York, NY, USA, 1997. ACM.
- [51] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018. USENIX Association.
- [52] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodík, Leana Golubchik, Minlan Yu, Victor Bahl, and Matthai Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018.

- [53] Ihab F Ilyas, Rahul Shah, Walid G Aref, Jeffrey Scott Vitter, and Ahmed K Elmagarmid. Rank-aware query optimization. In Proceedings of the 2004 ACM SIGMOD international conference on Management of data (ICMD), pages 203–214, 2004.
- [54] Samvit Jain, Junchen Jiang, Yuanhao Shu, Ganesh Ananthanarayanan, and Joseph Gonzalez. Rexcam: Resource-efficient, cross-camera video analytics at enterprise scale. CoRR, abs/1811.01268, 2018.
- [55] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: Scalable adaptation of video analytics. In Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), pages 253–266, New York, NY, USA, 2018. ACM.
- [56] Tian Jin and Seokin Hong. Split-cnn: Splitting window-based operations in convolutional neural networks for memory system optimization. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 835–847, 2019.
- [57] Christoph Käding, Erik Rodner, Alexander Freytag, and Joachim Denzler. Fine-tuning deep neural networks in continuous learning scenarios. In Chu-Song Chen, Jiwen Lu, and Kai-Kuang Ma, editors, Computer Vision – ACCV 2016 Workshops, pages 588–605, Cham, 2017. Springer International Publishing.
- [58] Daniel Kang, Peter Bailis, and Matei Zaharia. Blazet: Fast exploratory video queries using neural networks. arXiv preprint arXiv:1805.01046, 2018.
- [59] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: Optimizing neural network queries over video at scale. Proc. VLDB Endow., 10(11):1586–1597, August 2017.
- [60] K. Kang, W. Ouyang, H. Li, and X. Wang. Object detection from video tubelets with convolutional neural networks. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 817–825, June 2016.
- [61] Nick Koudas and Kenneth C Sevcik. High dimensional similarity joins: Algorithms and performance evaluation. IEEE Transactions on Knowledge and Data Engineering (TKDE), 12(1):3–18, 2000.
- [62] Sanjay Krishnan, Adam Dziedzic, and Aaron J. Elmore. Deeplens: Towards a visual data management system. In CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings, 2019.
- [63] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, Advances in Neural Information Processing Systems (NIPS), pages 1097–1105. Curran Associates, Inc., 2012.
- [64] Hei Law and Jia Deng. Cornernet: Detecting objects as paired keypoints. International Journal of Computer Vision (IJCV), Aug 2019.
- [65] Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. Reducto: On-camera filtering for resource-efficient real-time video analytics. 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 (SIGCOMM), pages 359–376, 2020.
- [66] Mike Liao. Benchmarking hardware for cnn inference in 2018. <https://towardsdatascience.com/benchmarking-hardware-for-cnn-inference-in-2018-1d58268de12a>, 2018.
- [67] Alan J Lipton, Peter L Venetianer, Niels Haering, Paul C Brewer, Weihong Yin, Zhong Zhang, Li Yu, Yongtong Hu, Gary W Myers, Andrew J Chosak, et al. Video analytics for retail business process monitoring, 2015. US Patent 9,158,975.
- [68] Mason Liu and Menglong Zhu. Mobile video object detection with temporally-aware feature maps. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018.
- [69] NIST. The spectrum crunch. <https://www.nist.gov/topics/advanced-communications/spectrum-crunch>, 2019.
- [70] Chrisma Pakha, Aakanksha Chowdhery, and Junchen Jiang. Reinventing video streaming for distributed vision analytics. In 10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18), Boston, MA, 2018. USENIX Association.
- [71] Niketan Pansare, Vinayak R Borkar, Chris Jermaine, and Tyson Condie. Online aggregation for large mapreduce jobs. Proceedings of the VLDB Endowment, 4(11):1135–1145, 2011.
- [72] Ziv Paz. Innovation in surveillance: What’s changing at the edge, core and cloud? <https://blog.westerndigital.com/innovation-surveillance-edge-core-cloud/>, year = 2018.

- [73] Alex Poms, Will Crichton, Pat Hanrahan, and Kayvon Fatahalian. Scanner: Efficient video analysis at scale. *ACM Trans. Graph.*, 37(4):138:1–138:13, July 2018.
- [74] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. *SIGCOMM Comput. Commun. Rev.*, 45(4):421–434, August 2015.
- [75] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S. Pai, and Michael J. Freedman. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 275–288, Seattle, WA, 2014. USENIX Association.
- [76] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen. Deepdecision: A mobile deep learning framework for edge video analytics. In *IEEE Conference on Computer Communications (INFOCOM)*, pages 1421–1429, April 2018.
- [77] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.
- [78] Venkatesh Saligrama and Zhu Chen. Video anomaly detection based on local statistical aggregates. *2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2112–2119, 2012.
- [79] Cosma Rohilla Shalizi. Advanced data analysis from an elementary point of view. <http://www.stat.cmu.edu/~cshalizi/ADAfaEPoV/ADAfaEPoV.pdf>, year = 2019.
- [80] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 322–337, 2019.
- [81] Haichen Shen, Seungyeop Han, Matthai Philipose, and Arvind Krishnamurthy. Fast video classification via adaptive cascading of deep models. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.
- [82] Honghui Shi. Geometry-aware traffic flow analysis by detection and tracking. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pages 116–120, 2018.
- [83] Ervin Teng, João Diogo Falcão, and Bob Iannucci. Clickbait: Click-based accelerated incremental training of convolutional neural networks. *CoRR*, abs/1709.05021, 2017.
- [84] Ervin Teng, Rui Huang, and Bob Iannucci. Clickbait-v2: Training an object detector in real-time. *CoRR*, abs/1803.10358, 2018.
- [85] Paul Viola, Michael Jones, et al. Rapid object detection using a boosted cascade of simple features. *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition (CVPR)*, 1:511–518, 2001.
- [86] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. CLARINET: Wan-aware optimization for analytics queries. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 435–450, Savannah, GA, 2016. USENIX Association.
- [87] Ashish Vulimiri, Carlo Curino, P. Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 323–336, Oakland, CA, 2015. USENIX Association.
- [88] Ashish Vulimiri, Carlo Curino, Philip Brighten Godfrey, Thomas Jungblut, Konstantinos Karanasos, Jitendra Padhye, and George Varghese. Wanalytics: Geo-distributed analytics for a data intensive world. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1087–1092, New York, NY, USA, 2015. ACM.
- [89] Haizhong Wang, Kimberly Rudy, Jia Li, and Daiheng Ni. Calculation of traffic flow breakdown probability to optimize link throughput. *Applied Mathematical Modelling*, 34(11):3376 – 3389, 2010.
- [90] Hao Wang and Baochun Li. Lube: Mitigating bottlenecks in wide area data analytics. In *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*, Santa Clara, CA, 2017. USENIX Association.
- [91] Junjue Wang, Ziqiang Feng, Zhuo Chen, Shilpa George, Mihir Bala, Padmanabhan Pillai, Shao-Wen Yang, and Mahadev Satyanarayanan. Bandwidth-efficient live video analytics for drones via edge computing. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 159–173, 2018.

- [92] Yiding Wang, Weiyan Wang, Junxue Zhang, Junchen Jiang, and Kai Chen. Bridging the edge-cloud barrier for real-time advanced vision analytics. In 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19), 2019.
- [93] Slate William Saletan. The case for mass surveillance. https://www.delcotimes.com/news/the-case-for-mass-surveillance/article_61a27a3c-8e8b-54e3-b048-e44682b6a024.html, 2013.
- [94] Mengwei Xu, Xiwen Zhang, Yunxin Liu, Gang Huang, Xuanzhe Liu, and Felix Xiaozhu Lin. Approximate query service on autonomous iot cameras. In Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services, pages 191–205, 2020.
- [95] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. Deepcache: Principled cache for mobile deep vision. In Proceedings of the 24th Annual International Conference on Mobile Computing and Networking, pages 129–144, 2018.
- [96] Tiantu Xu, Luis Materon Botelho, and Felix Xiaozhu Lin. Vstore: A data store for analytics on large videos. In Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys), pages 16:1–16:17, New York, NY, USA, 2019. ACM.
- [97] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li. Lavea: Latency-aware video analytics on edge computing platform. In 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pages 2573–2574, June 2017.
- [98] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. Awstream: Adaptive wide-area streaming analytics. In Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), pages 236–252, New York, NY, USA, 2018. ACM.
- [99] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 377–392, Boston, MA, 2017. USENIX Association.
- [100] Tan Zhang, Aakanksha Chowdhery, Paramvir (Victor) Bahl, Kyle Jamieson, and Suman Banerjee. The design and implementation of a wireless video surveillance system. In Proceedings of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom), pages 426–438, New York, NY, USA, 2015. ACM.
- [101] Hongwei Zhu, Farzin Aghdasi, Greg M Millar, and Stephen J Mitchell. Online learning method for people detection and counting for retail stores, 2017. US Patent 9,639,747.
- [102] Xizhou Zhu, Jifeng Dai, Lu Yuan, and Yichen Wei. Towards high performance video object detection. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 7210–7218. IEEE Computer Society, 2018.

ASAP: Fast Mobile Application Switch via Adaptive Prepaging

Sam Son[†] Seung Yul Lee[†] Yunho Jin[†] Jonghyun Bae[†] Jinkyu Jeong[‡] Tae Jun Ham[†]
Jae W. Lee[†] Hongil Yoon^{††}
[†]*Seoul National University* [‡]*Sungkyunkwan University* ^{††}*Google*

Abstract

With mobile applications' ever-increasing demands for memory capacity, along with a steady increase in the number of applications running concurrently, memory capacity is becoming a scarce resource on mobile devices. When the memory pressure is high, current mobile OSes often kill application processes that have not been used recently to reclaim memory space. This leads to a long delay when a user relaunches the killed application, which degrades the user experience. Even if this mechanism is disabled to utilize a compression-based in-memory swap mechanism, relaunching the application still incurs a substantial latency penalty as it requires the decompression of compressed anonymous pages and a stream of I/O accesses to retrieve file-backed pages into memory. This paper identifies conventional demand paging as the primary source of this inefficiency and proposes ASAP, a mechanism for fast application switch via adaptive prepaging on mobile devices. ASAP performs prepaging by combining i) high-precision switch footprint estimators for both file-backed and anonymous pages, and ii) efficient implementation of the prepaging mechanism to minimize resource waste for CPU cycles and disk bandwidth during an application switch. Our evaluation using eight real-world applications on Google Pixel 4 and Pixel 3a demonstrates that ASAP can reduce the switch time by 22.2% and 28.3% on average, respectively (with a maximum of 33.3% and 35.7%, respectively), over the vanilla Android 10.

1 Introduction

With the broad capabilities and flexibility of mobile computing, mobile applications continue to tout rich features to meet users' diverse demands. This entails a continuous increase in both codes and data footprint [4, 31]. This trend has resulted in a constant demand for larger memory capacity on mobile devices to address memory pressure issues. However, the cost of the device and the power/area budget often limit its size.

Modern mobile OSes support virtual memory with compression-based swap [3, 13]. The virtual memory pro-

vides an illusion of physical memory space larger than the actual memory capacity, enabling multiple applications to run concurrently even under high memory pressure. However, the benefits come with additional overhead, degrading performance. Slow I/O accesses increase the latency of fetching non-resident file-backed pages from storage. To fetch anonymous pages in the compressed swap space, they first need to be decompressed by CPUs at a page fault. Allocating free pages also consumes system resources. Fetching pages on-demand via demand paging may not efficiently utilize available resources such as CPU cycles and I/O bandwidth.

Our empirical analysis shows that the application switch time can increase by a factor of $4\times$ (in the order of hundreds of milliseconds) when the system is experiencing memory pressure, possibly when running many background applications. This slowdown is mainly attributed to the long blocking time introduced by demand paging for both file-backed and anonymous pages during the application switch rather than freeing allocated pages.

A recent study shows that today's smartphone users often run more than 10 applications [25], and thus it is likely that the system is often operating under memory pressure unless the phone has a large main memory capacity. It is also known that users switch between applications more than 100 times per day [9]. We speculate that such frequent, long-latency events can potentially affect smartphone user experience negatively. In this paper, we aim to reduce the latency of the application switch by minimizing the demand-paging related slowdown. To achieve this goal, we propose ASAP, a mechanism for fast application switch via adaptive prepaging. ASAP builds on the following key observations:

- Hardware resources for fetching non-resident pages are underutilized during the application switch when the system is under memory pressure. For eight popular Android applications, CPU utilization is 34.2% during the switch. Also, only 19.4% of the maximum disk bandwidth is used on average.
- File-backed pages and anonymous pages have differ-

ent characteristics in their *switch footprint*, a set of accessed pages during the application switch. In particular, the switch footprint for file-backed pages is much more invariant—about 75% of all accessed file-backed pages are invariant across switches, while only 44% of anonymous pages are invariant. This motivates us to use different prediction strategies for prepagating them.

We capitalize on these empirical observations to develop an effective prepagating approach. The first observation suggests that it is promising to utilize available resources to prepage pages likely to be accessed at the beginning of an application switch. The prepagating is helpful to maximize the effective CPU and disk bandwidth utilization, which can translate to performance gains (i.e., reduced switch time). The second observation suggests that the target pages to fetch need to be adapted at runtime to capture the applications' dynamically changing page access patterns. This improves the prediction accuracy for the switch footprint, hence making ASAP more effective.

At an application switch, ASAP wakes up multiple prepagating threads to start fetching both file-backed pages and anonymous pages. These threads run in parallel with application threads to overlap prepagating with application computation. To accurately predict switch footprint pages, ASAP employs an adaptive prediction mechanism. Specifically, a single predictor maintains two tables: a candidate table and a target table. The predictor promotes or demotes pages between the two tables based on the runtime information of their access patterns. The prepagating threads issue fetch requests only for the pages in the target table, while pages having a smaller chance of being accessed are maintained in the candidate table.

We have implemented ASAP in Android OS and evaluated it using a set of eight popular mobile applications on Google Pixel 4 and Pixel 3a. The evaluation results show that ASAP considerably reduces the application switch time under memory pressure. ASAP reduces the switching time by 22.2% and 28.3% on average (33.3% and 35.7% at maximum) on Pixel 4 and Pixel 3a, respectively, over the vanilla Android 10. This improvement is attributed to an average of 39.8% and 25.2% increase in CPU and disk bandwidth utilization, respectively, as well as 79.3% and 68.4% prediction accuracy for file-backed and anonymous pages, respectively.

In summary, this paper makes the following contributions:

- We empirically analyze the performance bottleneck of the application switch to identify opportunities for prepagating as a solution to the problem.
- We propose ASAP, an adaptive prepagating technique to reduce the switch time, which is a key user interaction on the mobile device. ASAP is application-agnostic without requiring any change to application codes.

- We integrate ASAP into Android OS and evaluate its performance by using eight popular mobile applications on high-end and mid-end devices (Google Pixel 4 and Pixel 3a). The results demonstrate the effectiveness of ASAP for reducing the application switch time by 22.2% and 28.3% on average, respectively, over the vanilla Android 10.

2 Background and Motivation

2.1 Android Application Memory Management

Application Lifecycle and Memory Management. In Android OS, an application (specifically the application activity) is either in the foreground (i.e., having focus) or in the background (e.g., not visible). In other words, the application that a user is actively using is considered to be in the foreground, while the applications that have been launched but are not currently being used are considered to be in the background. When the system has sufficient DRAM, all application data are kept in memory. However, a user often utilizes many different applications over time, and eventually, application data exceed the DRAM capacity. In such a case, the Android low memory killer daemon (*lmkd*) identifies the least essential application (e.g., one in the background) and kills it so that the memory space that it occupied is freed [21, 26]. Note that this does not necessarily result in the complete loss of the application state since Android applications often store a minimal set of its states when the application is moved to the background. With this mechanism, Android OS only stores a small set of essential application data in memory. For this reason, when a user starts an application that was moved to the background a long time ago and hence killed by *lmkd*, the application data is not resident in memory. Instead, the application needs to recreate all of its activities from scratch utilizing the saved state information. On the other hand, when a user starts an application that was moved to the background very recently, it is much more likely that this application's data still resides in memory, and the application will be ready-to-use in a much shorter period. The time Android OS requires for the former case is called *launch time* and the latter is called *switch time* (sometimes also called *hot launch time*).

Compression-based Swap. An alternative approach to secure the free memory space is the compression-based swap, which compresses the least essential memory pages and stores them in a separate memory region. Later, when the application accesses the compressed pages, they are decompressed back to memory. Compared to the traditional disk-based swap mechanism, the compression-based in-memory swap is faster since it can avoid long-latency disk accesses. This approach's drawback is that i) compressed memory pages still consume memory capacity, and ii) compression/decompression spends CPU cycles. This mechanism is enabled by

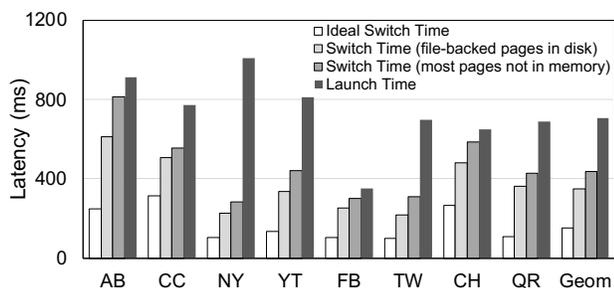


Figure 1: Application switch latency across different scenarios. *Ideal Switch Time* represents a case where all of the applications’ anonymous and file-backed pages reside in memory. *Switch Time (file-backed pages in disk)* represents a case where all of the applications’ anonymous pages are resident in memory, but almost all of file-backed pages do not reside in memory. *Switch Time (most pages not in memory)* represents a case where most of the applications’ anonymous pages are already swapped out, so they are compressed and stored in the compressed memory pool based on the compression-based in-memory swap. Also, almost all of file-backed pages are evicted from memory due to memory pressure. This latency information is equivalent to the baseline switch time illustrated in Section 6.1. Finally, *Launch Time* indicates a case where an application needs to start from scratch. Refer to Section 6.1 for the detailed methodology.

default in many commercial mobile OSes such as Android OS and Apple iOS [3, 13]. However, in practice, Android OS by default does not actively utilize this mechanism since *lmkd* is often triggered first to reclaim memory space before a swap happens [21, 25, 26, 35].

2.2 Launch Time and Switch Time

When a user relaunches an application after a while since its last usage, the latency to reload may differ depending on the system’s memory pressure. For example, if the system’s memory pressure is low (e.g., the system has not used much memory since the application’s last launch), the application’s data will still reside in memory. Thus the application could reload quite quickly (i.e., *ideal switch time*). On the other hand, if the system’s memory pressure is high (e.g., the user utilized many different apps during the time window), the application will be killed by *lmkd*. The relaunch is highly likely to require recreating the application’s activities, incurring a much longer delay (i.e., *launch time*). Finally, if *lmkd* is disabled, the compression-based swap mechanism will come into play. The application’s anonymous pages will be stored in memory in a compressed form, and the file-backed pages will be discarded. In this case, relaunching an application requires decompressing some of the application’s anonymous pages and reloading file-backed pages from the disk.

Figure 1 presents the application launch/switch time of eight real-world applications (AB: Angry Bird, CC: Candy Crush Saga, NY: New York Times, YT: YouTube, FB: Facebook, TW: Twitter, CH: Google Chrome, QR: Quora) on Google Pixel 4. The figure shows that the switch time is lower than the launch time in all applications. This indicates that reconstructing activities of an application from scratch requires more time than retrieving the relevant anonymous pages and file-backed pages from memory and disks, respectively. This implies that an aggressive setting of Android *lmkd* increases the time to relaunch the application, which confirms the findings of the previous literature [18, 20, 21]. To avoid this unnecessary delay in relaunching the application, it is better to lower the *lmkd* threshold (or even disable it) so that the system can utilize compression-based swap more actively.

This figure also shows a significant gap between the ideal switch time and the switch times under memory pressure. The gap between the *ideal switch time* and the *switch time (file-backed pages in disk)* quantifies the overhead of retrieving file-backed pages from the disk. The gap between the *switch time (file-backed pages in disk)* and the *switch time (most pages not in memory)* indicates the overhead of decompressing anonymous pages from the compressed memory pool. In fact, this overhead increases the application switch time by a factor of 4× relative to the ideal switch time on average. Unfortunately, the real-world switch time is often closer to the switch time (most pages not in memory) than the ideal switch time when we consider recent trends: i) an increase in an application’s memory capacity requirements and ii) an increase in the number of apps that a user runs concurrently.

To the best of our knowledge, there are no concrete studies on the threshold of user perception of the application switch. However, previous studies [7, 27] on related contexts imply that the delay of hundreds of milliseconds in the application switch may degrade the user experience. According to Card [7], users feel that a system is reacting instantaneously only when the response time is shorter than 100 ms. Olen-ski [27] reports that a 100 ms delay in web page loading can degrade the user experience, resulting in a 1% drop in a company’s revenue. Thus, we are convinced that maintaining a lower switch time over a wide variety of usage scenarios is critical for user experience.

2.3 Opportunities for Prepaging

Limitations of Demand-Paging. Figure 1 shows that the switch time under memory pressure is substantially worse than the ideal switch time. We find that such a huge overhead resulting from i) decompressing anonymous pages and ii) retrieving file-backed pages from the disk is attributable to the inefficiencies of demand paging. Figure 2(a) shows the CPU utilization and Figure 2(b) shows the disk bandwidth utilization of Google Pixel 4 during switch time of eight applications under memory pressure. Overall, the CPU utilization

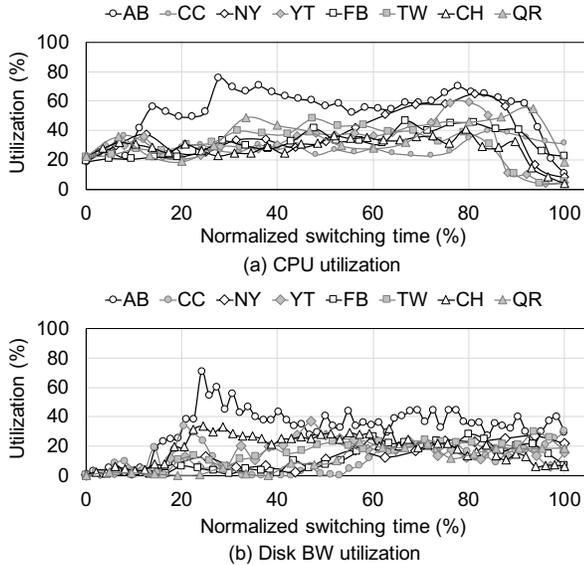


Figure 2: CPU and disk bandwidth utilization of a high-end device (Pixel 4) during the switch time.

remains relatively low (i.e., less than 50%) for all applications except AB. Similarly, disk bandwidth utilization is also much lower than the sustainable peak bandwidth most of the time. As shown in Figure 3 for Google Pixel 3a, its disk bandwidth utilization is higher than that of the high-end device (Pixel 4) because the disk bandwidth is relatively low. However, the empirical results show that the resources are not still fully utilized in both cases.

Ideally, the CPU should have been fully utilized to decompress compressed memory pages, and disk bandwidth should have been saturated to retrieve file-backed pages from the disk. However, since the default demand paging mechanism initiates the decompression of memory pages and I/O accesses only at a page fault, the system wastes available resources and spends more application switch time than necessary.

Opportunities and Challenges of Prepaging. The key idea behind ASAP is that we can significantly improve the switch time by letting prepaging threads aggressively decompress memory pages and perform I/O accesses before the application codes demand them. By doing so, ASAP can fully exploit the available system resources (i.e., CPU cycles and disk bandwidth), making the switch time under memory pressure much closer to the ideal switch time. There are two main challenges in this approach. First, the system should effectively identify the switch footprint, a set of pages to be accessed during the switch. Second, the prepaging threads should be efficiently implemented to fully exploit available resources while minimizing their interference with application threads. The following sections describe how ASAP addresses these two challenges.

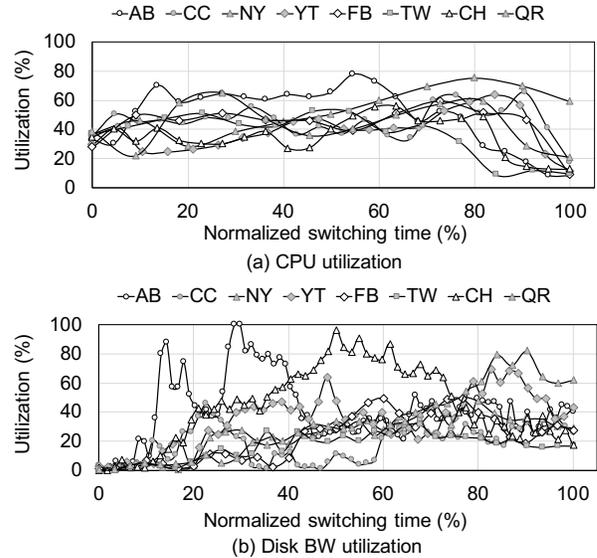


Figure 3: CPU and disk bandwidth utilization of a mid-end device (Pixel 3a) during the switch time.

3 ASAP Design Overview

The empirical observations in the previous section suggest that it is promising to design an adaptive prepaging. We first set two key requirements to design a practical prepaging mechanism:

1. The proposed design should be able to accurately predict a set of pages that are likely to be accessed during an application’s switch time (i.e., switch footprint).
2. The proposed design should be able to maximize the efficiency of prepaging by achieving high system resource utilization (i.e., CPU cycles and disk bandwidth) without interfering with the execution of application threads.

ASAP satisfies these requirements with *Switch Footprint Estimator (SFE)* and *Prepaging Manager*. We have integrated them into the application switching process in Linux kernel. Thus, ASAP is application-agnostic without requiring any changes to application codes.

Figure 4 illustrates the overall structure of ASAP with key components shaded in gray. SFE consists of two estimators: one for anonymous pages and the other for file-backed pages. Based on the analysis on the switch footprint, SFE for file-backed pages utilizes offline profiling results as well as a lightweight runtime module to estimate the mostly invariant switch footprint of file-backed pages. On the other hand, SFE for anonymous pages is designed to track a dynamic switch footprint of anonymous pages by gradually promoting pages that are likely to be fetched again during the next switch.

These estimators generate a set of target pages for prepaging, which are retrieved at the beginning of an application

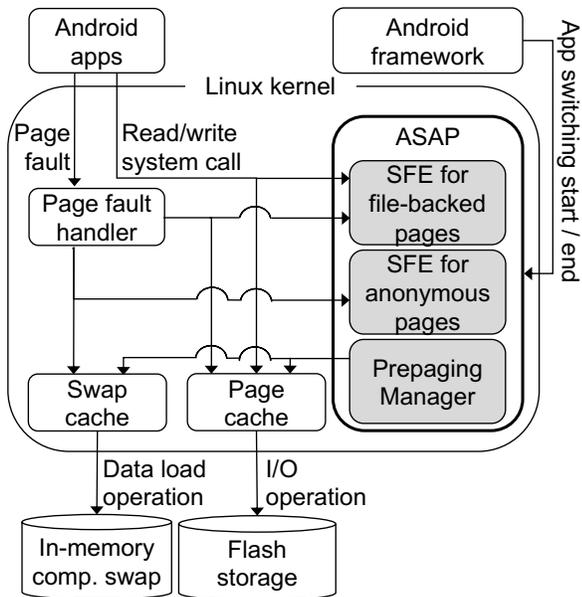


Figure 4: ASAP design overview.

switch. It is possible that page information in the prepping target table becomes obsolete due to inconsistent memory reuse patterns of applications (e.g., application update, post-installation optimization with *dexopt*). Based on the hit/miss history of prepping, the information is updated over time to ensure high prediction accuracy.

Preparing Manager is responsible for prepping threads that are used to fetch target pages from a prepping target table. It monitors a timing signal that notifies the start and the end of the application switch event from the Android framework. Prepping Manager promptly wakes up inactive prepping threads for the switched application when it receives a start signal for application switch, and then it initiates prepping. Multiple prepping threads are created according to the number of available CPU cores and run in parallel with application threads to fully utilize the available system resources such as CPU cycles and disk bandwidth. Once they finish issuing fetch requests for all the pages from the prepping target table, the prepping manager makes them sleep again until the next switch.

4 Switch Footprint Estimator

4.1 Switch Footprint Analysis

To effectively estimate the targets for prepping, it is important to understand the characteristics of the switch footprint: a set of pages that are accessed during the switch time. For this purpose, we perform an experiment that exhaustively records all pages accessed across 10 switches for each application

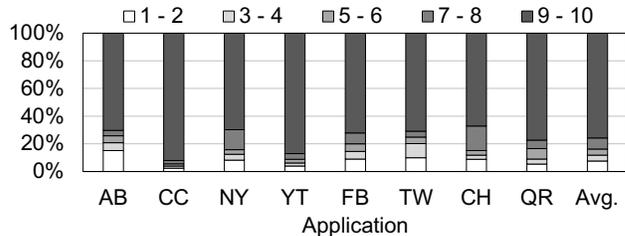


Figure 5: Switch locality analysis for file-backed pages.

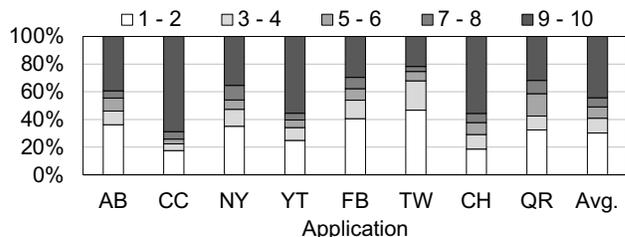


Figure 6: Switch locality analysis for anonymous pages.

(experimental details are available in Section 6.1). For this we cleared the access bit of all present PTE in the address space just before the switch and then checked them right after the switch is completed.

File-backed Pages. Figure 5 shows the switch footprint composition for file-backed pages. The stacked bar shows how many times pages are accessed over the 10 application switches. The switch footprint is largely invariant in this case. On average, about 75% of pages are accessed 9 or 10 times and only 10% are accessed fewer than five times. This highly invariant access pattern of the file-backed pages is due to the fact that a large part of codes and shared library files keep being loaded for the execution of an application.

Anonymous Pages. Figure 6 shows the switch footprint composition for anonymous pages. The access pattern is not as invariant as file-backed pages. About 44% of the anonymous pages are accessed 9 or 10 times across 10 switches. The portion of the invariant (i.e., always accessed) pages is much smaller as the set of accessed anonymous pages easily changes when the application context changes.

Implications. As the characteristics of the switch footprint for anonymous pages and file-backed pages differ, so should their switch footprint estimators. Estimation for file-backed pages can exploit the fact that file-backed pages are highly invariant to minimize the runtime overhead. On the other hand, estimation for anonymous pages needs to rely more on the runtime information so that it can correctly track dynamically changing switch footprints across switch events. Still, the runtime overhead of tracking the switch footprint for anonymous pages is relatively low as the number of anonymous pages in the switch footprint is much smaller than that of the file-backed pages, as shown in Figure 7. The rest of this section discusses the SFE design for both file-backed pages (Section 4.2) and anonymous pages (Section 4.3).

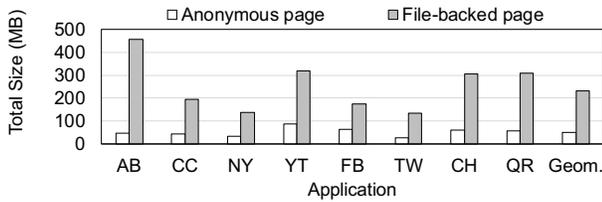


Figure 7: Switch footprint of anonymous and file-backed pages across different applications.

4.2 Estimator for File-Backed Pages

As shown in Figure 5, a major portion of file-backed pages accessed during the application switch are invariant across switches. To exploit this characteristic, SFE for file-backed pages first performs offline profiling to identify the set of potential candidates for prepagging, and then later utilizes minimal runtime information to maintain a concise set of prepagging targets, as shown in Figure 8.

Offline Profiling. The estimator performs offline profiling to obtain a set of prepagging candidates. For this purpose, we measure the file-backed pages that are accessed during ten switch events for each app, as in Figure 5. Then, pages accessed more than eight times (out of ten trials) are considered to be frequently accessed. The resulting set of pages is stored as a file (*Offline Candidate Table*). Specifically, as shown in Figure 8, the profiled result is stored as a map, where a filename is a key and a list of pairs (offset, len) is a value. Each pair represents [offset, offset + len) pages within a file that are accessed during an application switch (we call it an *extent* in the rest of this paper). Later, the profiled result is reloaded at the launch time of this application.

Fault Logging. Fault logging happens at every switch event. Specifically, SFE logs the inode and page indices of all faulted extents received from the kernel until the end of the switch time. This is stored in a *fault buffer*, which is later utilized by the estimator after the end of the switch time to update its prepagging targets.

Prepagging Target Management - Insertion. Once the switch finishes, a background thread performs prepagging target management, exploiting the information from the offline profiling and the fault logging. *Prepagging Target Table* stores information for extents that are to be fetched by the prepagging threads. Ideally, we should insert only those extents that are likely to be fetched in the near future. To identify such an extent, the estimator first inspects an extent in the fault buffer and checks if the extent is also found in the *Offline Candidate Table*. If so, the estimator inserts the corresponding entries into the *Prepagging Target Table*.

Prepagging Target Management - Extent Merging. The *Prepagging Target Table* may have multiple extents on the same file. In such a case, if two extents are close to each other (e.g., the end of one extent is less than 16 pages apart from

the start of the other extent), we merge those two extents and create a larger extent that covers both. This is to avoid issuing multiple fragmented I/O requests and instead issue a single, sequential large I/O request, which is often handled much more efficiently.

Prepagging Target Management - Eviction. Eviction from a *Prepagging Target Table* happens when the fetched page turns out to be not utilized during a switch time. Specifically, the estimator checks the mapcount of each fetched page after the switch, and removes the page from the *Prepagging Target Table* if the mapcount is zero. When a page is part of an extent, the extent is divided into two smaller extents.

4.3 Estimator for Anonymous Pages

As shown in Figure 6, the set of anonymous pages accessed during the application switch changes much more frequently than files. Moreover, anonymous pages are allocated whenever the application is launched, and thus offline profiling is not helpful for identifying prepagging candidates. To effectively track the switch footprint for anonymous pages, we focus on runtime analysis, unlike the case of file-backed pages (Section 4.2). Policies of the estimator are depicted in Figure 9.

Fault Logging. During the application switch time, like the estimator for file-backed pages, the Switch Footprint estimator for anonymous pages logs all anonymous page faults. Fault information is logged at a fault buffer for later usage.

Access Logging. To track access information during switch time, this estimator clears the access bit of every PTE represented by each page identifier in both the *Prepagging Target Table* and the *Online Candidate Table* before every application switch time. Then, right before the end of the switch time, the access bits of all pages in both tables are again checked to identify a set of pages that are accessed during switch time.

Prepagging Target Management - Check & Insertion. After the application switch time, this estimator first checks if each page in the fault buffer is not already present in the *Online Candidate Table* nor the *Prepagging Target Table*. If there are pages that are not already present in the tables, they are inserted into the *Online Candidate Table*.

Prepagging Target Management - Promotion. Also, the estimator checks if each page in *Online Candidate Table* has been accessed during the switch time by inspecting the access log. If a page has been accessed during the last switch time, the page in the *Online Candidate Table* is then promoted to the *Prepagging Target Table*.

Prepagging Target Management - Eviction. Every page in both the *Online Candidate Table* and the *Prepagging Target Table* has its own timeout counter, which is the number of switch events a page can experience before getting evicted from a table. The timeout counter (e.g., 5) of a page is decremented after every switch time. If a specific page is not accessed until the timeout counter reaches zero, it is evicted from the

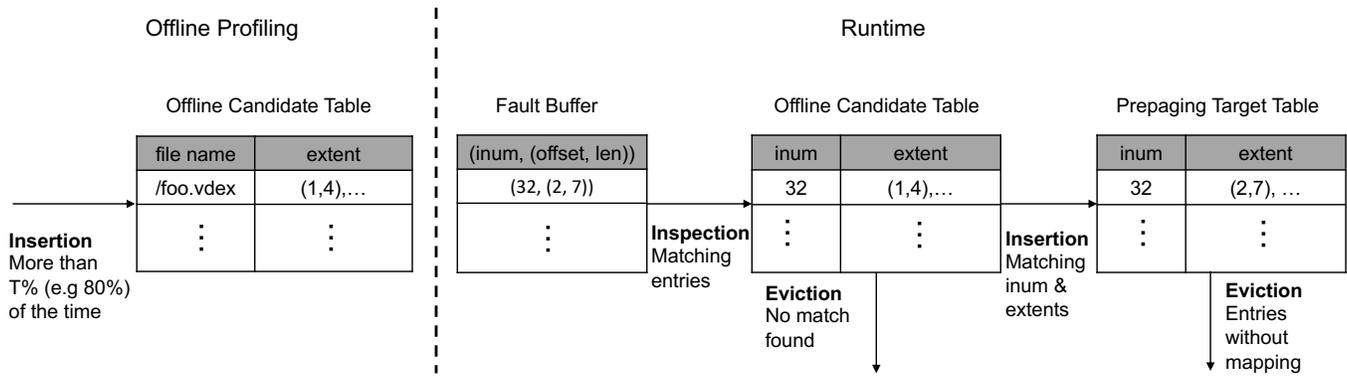


Figure 8: Switch footprint estimator for file-backed pages.

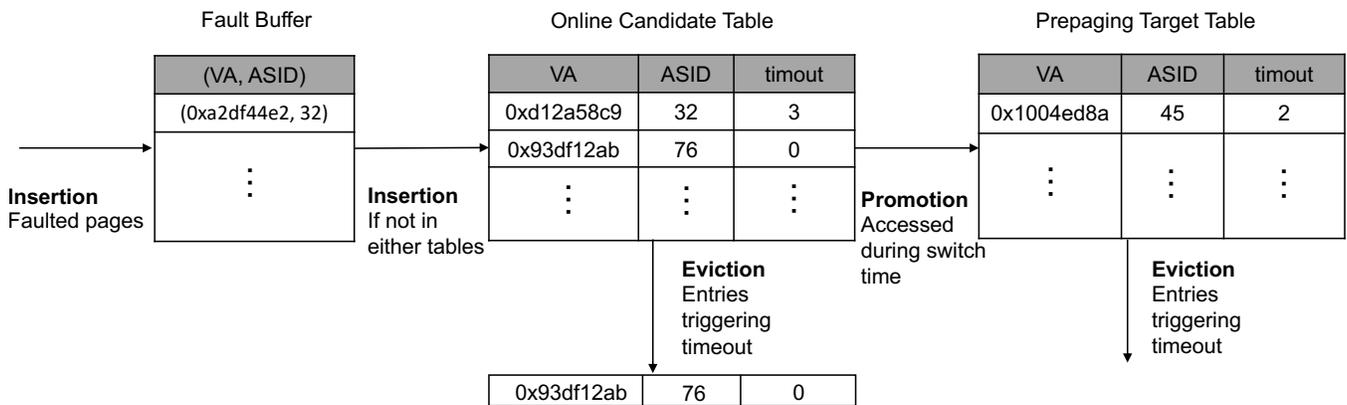


Figure 9: Switch footprint estimator for anonymous pages.

table that it belongs to. But, whenever a page is accessed, the timeout counter of an identifier is reset to the default timeout counter value (e.g., 5).

5 Prepaging Manager

Whenever an application switch event occurs, ASAP’s Prepaging Manager wakes up threads. They prefetch pages in the *Prepaging Target Table*, which eventually constructs corresponding PTEs. We apply different prepaging policies to anonymous pages and file-backed pages as follows.

5.1 Prepaging Anonymous Pages

Prepaging of anonymous pages requires decompressing swapped out pages in the compressed in-memory swap space. Hence, the task is compute-intensive, and the task should be carefully scheduled not to incur CPU contention between application threads and prepaging threads. Although the CPU utilization is low, as we reported in Section 2.3, the application threads can demand more CPU resources due to reduced

page fault events by prepaging operations.

To this end, the prepaging manager maintains a set of threads for anonymous page prepaging. We pinned a thread on each core, and we assigned the lowest priority (i.e., SCHED_IDLE [17]) to them. This allows us to opportunistically utilize the surplus CPU resources for the prepaging of anonymous pages while not incurring any CPU contention with the application threads.

The distribution of prepaging work is done in a work sharing manner. Each thread retrieves a batch (16 pages) from the *Prepaging Target Table*, and then conducts the prepaging operations for pages in the batch. Specifically, for each virtual page in the batch, each thread checks whether the virtual page is present in the process’s address space. If non-present, it issues a swap-in operation for the virtual page to the swap subsystem (i.e., the swap cache). The swap-in operation eventually becomes the decompression operation in the in-memory compressed swap device. After the target page is decompressed, the thread finally makes the corresponding PTE point to the swapped-in page.

5.2 Preparing File-backed Pages

The prepaging manager maintains another set of threads for prepaging file-backed pages. However, file-backed pages impose a higher miss penalty than anonymous pages due to long disk I/O time. Therefore, we take a different prepaging policy for file-backed pages as follows.

First, a file is a unit of prepaging work distribution. Each thread is assigned a file from the *Prepaging Target Table*, and then prefetches pages of the file. For the prepaging operation, each thread issues asynchronous page cache read operations for the corresponding extents in the *Prepaging Target Table*. Note that the *Prepaging Target Table* estimates pages are not only accessed through page faults but are also accessed via read/write system calls. Hence, not all prefetched pages need to be mapped in the process’s virtual address space. Prepaging the pages in the *Prepaging Target Table* places fetched pages into the page cache, and thus page faults can still occur for those prefetched pages. However, their fault handling cost is light when the corresponding pages are in the page cache (i.e., minor faults in Linux). When a page fault occurs for a file-backed page, the page fault handler retrieves pages surrounding the missing page from the page cache and maps them all together in the virtual memory to reduce page fault frequency; hence performing prepaging.

Second, we dedicate at least one thread to continuously perform prepaging of large file-backed pages. Figure 10 shows the cumulative distribution of accessed pages of files in the switch footprint of the applications. As shown in the figure, about 90% of the total number of accessed pages is part of the top 15% of large files. Thus, we can expect spatial locality of such accesses to large files. This indicates that if those files are assigned to the prefetching threads with the `SCHED_IDLE` priority, pages of the files are not likely to be prefetched on time due to the lowest CPU scheduling priority. To avoid this problem, we designate one thread with `SCHED_NORMAL` priority running on the big core to be in charge of prefetching pages of large files. Considering the big-LITTLE heterogeneity of the CPU cores in mobile systems, the thread is assigned to run on a big core to maximize the prefetching performance. We have empirically found that this configuration is effective in reducing the miss ratio as well as the CPU contention with application threads.

Lastly, during the prefetching of file-backed pages, file metadata should be carefully handled. Unless file metadata (e.g., logical block addresses of file pages) is in memory, the metadata retrieval incurs additional delay, which in turn degrades the effectiveness of prefetching [23]. This problem exacerbates because of our extent-based prefetching. The metadata read I/O can stay behind a large prefetching I/O request, thereby blocking the prefetching threads as well as the application threads. To avoid this problem, our scheme attempts to read metadata blocks before prefetching pages of a corresponding file. The metadata block reads are done

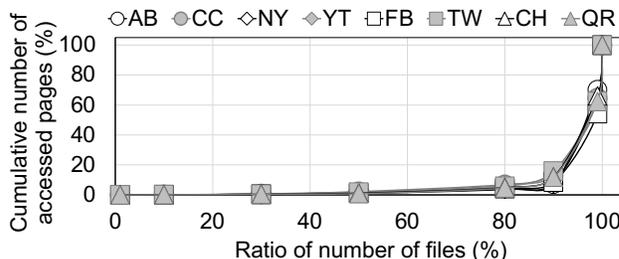


Figure 10: Cumulative number of accessed pages CDF of files across the various applications during switch from one application to another one. Files are sorted by size. 100% indicates the largest file.

by accessing file pages with a large stride in file offset, 512 pages in our case, because a direct block contains LBAs of 512 data blocks in F2FS [22].

6 Evaluation

In this section, we evaluate the effectiveness of ASAP. Section 6.1 describes the evaluation methodology and workloads. Then, we evaluate the latency benefits of our proposal in Section 6.2. Section 6.3 analyzes the accuracy of the switch footprint estimator. We evaluate the efficacy of the prepaging manager by considering improvement of the effective disk bandwidth and CPU utilization.

6.1 Methodology

Switching Latency Measurement. To measure the application switching latency, we used the `am` command in the Android debug bridge (`adb`) [2]. This command starts a selected application and reports two types of switching latency. One is latency from a user’s touch to the first rendering, and the other one is latency from a user’s touch to the full rendering [5]. The latter is reported only when the application developer implements the debug callback. The information is reported only by the YT application among eight benchmark applications. Thus, we use the time to the initial rendering as a metric. For the YT application, we observe that the additional latency overhead of the full rendering is less than 5% of the switch latency (10-20 ms). Users could also start to interact with applications in the middle of the rendering [16]. The actual latency overhead is expected to be insignificant when the performance benefits of ASAP are considered. This justifies our usage of the time to the initial rendering as the metric for evaluation.

System Configuration. For our evaluation, we use Google Pixel 4 and Pixel 3a, which represent high-end and mid-end devices, respectively. Table 1 describes their specifications. We implement ASAP in Android 10. When measuring the application switch overhead under memory pressure, we consider two aspects for our experimental methodology.

Table 1: Device Specifications

Device	Google Pixel 4	Google Pixel 3a
CPU	Octa-core Qualcomm Snapdragon 855	Octa-core Qualcomm Snapdragon 670
DRAM	6GB LPDDR4x (eff. 4GB)	4GB LPDDR4x
Storage	64GB UFS 2.1	64GB eMMC 5.1
OS	Android 10.0.0 (r41) with Linux kernel 4.14	Android 10.0.0 (r41) with Linux kernel 4.9
zram	2GB (default)	2GB (default)

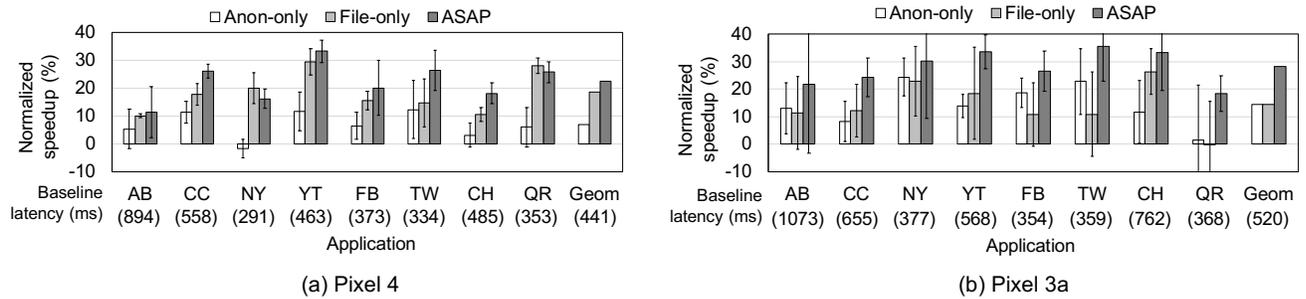


Figure 11: Normalized speedup of application switching latency on (a) Pixel 4 and (b) Pixel 3a. Numbers in parentheses indicate absolute switching latency of the baseline system in ms. Error bar shows standard deviation over different sequences.

Table 2: Applications and automated interactions to change contexts.

Application	Automated Usage Patterns
Angry Bird (AB)	Play a stage
Candy Crush (CC)	Play a stage
New York Times (NY)	Browse and read articles
Youtube (YT)	Watch videos
Facebook (FB)	Browse and read posts
Twitter (TW)	Browse and read posts
Chrome (CH)	Browse keywords
Quora (QR)	Browse questions and answers

Table 3: Chosen 3 application test sequences.

Sequence 1	YT-CH-CC-AB-NY-QR-FB-TW
Sequence 2	QR-NY-CH-CC-YT-TW-FB-AB
Sequence 3	AB-FB-QR-TW-CC-CH-YT-NY

First, we favor the compression based swap approach over the *lmkd*, which often acts first to secure free memory and prevents the system from being under memory pressure. Note that Android currently enables both features by default. We disable the *lmkd* for our evaluation to solely analyze the performance impact on application switch under memory pressure.

Second, users show different application usage patterns such as a spectrum of day-to-day use applications and the use of multitasking features. These lead to different memory usage patterns even among smartphone users who have the same devices.

In this work, thus, we focus on evaluating the memory

pressure impact of the application switch for a fixed set of a wide spectrum of top rated applications (refer to Table 2). We enable memory ballooning by considering the entire footprint of the target applications instead of enabling numerous applications to cause memory pressure for the target devices. The effective memory size of both Pixel 4 and Pixel 3a is 4GB. Throughout our evaluation, we refer to the switch time measured on this configuration as the *baseline switch time*.

Workloads and Automation of Tests. In order to reduce the run-to-run variation in the experimental results, we carefully devise an automation program that closely mimics a set of pre-determined user interactions with *adb*. For example, the Facebook (FB) usage pattern contains scrolling down the main news feed, searching for user profiles, and watching their timelines. Another example would be YouTube (YT), where our program searches and watches different video clips. The details of the usage patterns are listed in Table 2.

After execution of a certain application, e.g., Candy Crush (CC), we switch to the next application, e.g., TW, by following a pre-determined sequence of applications. As there are $8!$ available application sequences for eight applications, we chose three random sequences to evaluate ASAP (Table 3). The start and end time of the application switching operation is informed by the Android activity manager [1, 12]. We iterate the selected sequence 10 times and measure the application switch time. With this user interaction automation program, we repetitively conduct the same evaluation process.

6.2 Application Switch Latency

Figure 11 presents the speedup of ASAP based on Pixel 4 and Pixel 3a, respectively, for 8 applications. We also evaluate the speedup by selectively enabling prepaging for either anonymous pages or file-backed pages. Compared to the baseline, ASAP shows an average of 22.2% and 28.3% performance improvement, and a maximum of 33.3% (YT) and 35.7% (TW) on Pixel 4 and Pixel 3a, respectively. We observe 6.8% and 14.6% performance improvement on each device on average when ASAP performs prepaging only for anonymous pages (*Anon-only*). Among the eight applications, YT and TW show the most noticeable latency reduction on Pixel 4 and Pixel 3a, respectively. With prepaging for file-backed page only (*File-only*), the latency is reduced by 18.3% and 14.4% on average. Here, YT and CH show a substantial latency reduction on each device.

When both SFEs are enabled (ASAP), we observe additional performance benefits for most cases as expected. However, in NY and QR on Pixel 4, integrating both SFEs does not further reduce their switch latency.

6.3 Estimator Efficiency

Figure 12 presents the efficiency of the proposed switch footprint estimators for both Anonymous SFE and File-backed SFE. Since we observe similar performance trends on both devices, we will only present the results on Pixel 4 in the rest of this section. Precision is defined as a fraction of correctly prepaged pages among entire prepaged pages. Recall is defined as a fraction of correctly prepaged pages among all faulting pages during the switch time when the baseline system is considered. Anonymous SFE shows an average of 68.4% precision and 60.4% recall. File-backed SFE shows an average of 79.3% precision and 52.2% recall. The Switch Footprint Estimator for file-backed pages shows better precision relative to that of the Switch Footprint Estimator for anonymous pages. The difference comes from the fact that the switch footprint of file-backed pages is more static, as described in Figure 5.

The gap between precision and recall comes from the coverage of the prepaging target tables. Note that both precision and recall have the same numerator value while the denominator of recall can cover more pages that have not been fetched by the proposed prepaging scheme. We see a larger gap between the precision and the recall of file-backed page prepaging relative to the gap of anonymous page prepaging. This could result from the limited coverage of the candidate pages that is based on the static profiling. The static profiling approach may not capture the entire set of pages that are likely to cause faults at runtime.

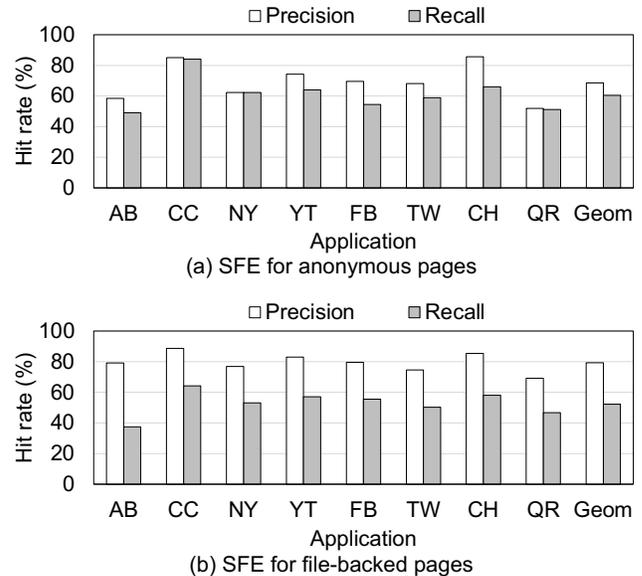


Figure 12: Switch footprint estimator performance.

6.4 Resource Utilization

To show the efficacy of ASAP on prepaging, we evaluated the changes in CPU and memory bandwidth utilization on Pixel 4. The bandwidth utilization is computed as the ratio of achieved file read throughput to the maximum sequential throughput measured in fio [10]. As depicted in Figure 13, ASAP eagerly allocates threads for decompression, which increases the CPU utilization to $1.18\times$ on average over the total switch time, compared to the baseline switch. We also notice the maximum of $1.35\times$ utilization increase. The CPU has been under utilized at the beginning of the application switch. In most cases, the anonymous prepaging threads have a large window of opportunity to fully exploit the available CPU resources. Therefore, when ASAP is enabled, the CPU utilization improves at the early stages of switching. Because the throughput of zram actually scales depending on the number of CPU cores, the anonymous prepaging threads can prepage anonymous pages at great speed. For most applications, prepaging threads finish at around the first 30% of normalized switching time. After that, the CPU utilization follows the CPU utilization pattern of the baseline. On the same page as the CPU, ASAP also improves the I/O bandwidth by 25.2% on average, as shown in Figure 14. In most cases, we observe a noticeable increase in the I/O bandwidth at the early stages of switching and the maximum achieved bandwidth is also higher than that of the baseline. ASAP does not induce significant improvement over the baseline in the case of AB. This is because AB is a highly parallel application with high I/O utilization. Therefore, the I/O bandwidth improvement from our asynchronous I/O threads is limited. The empirical analysis substantiates that ASAP efficaciously exploits the resources at the beginning of the switch to considerably reduce

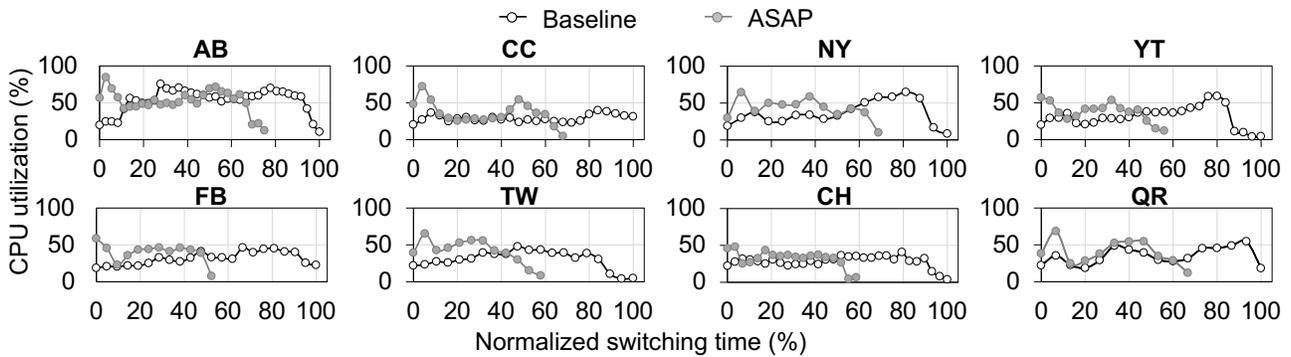


Figure 13: CPU utilization. X-axis is a timeline normalized to baseline’s switch time.

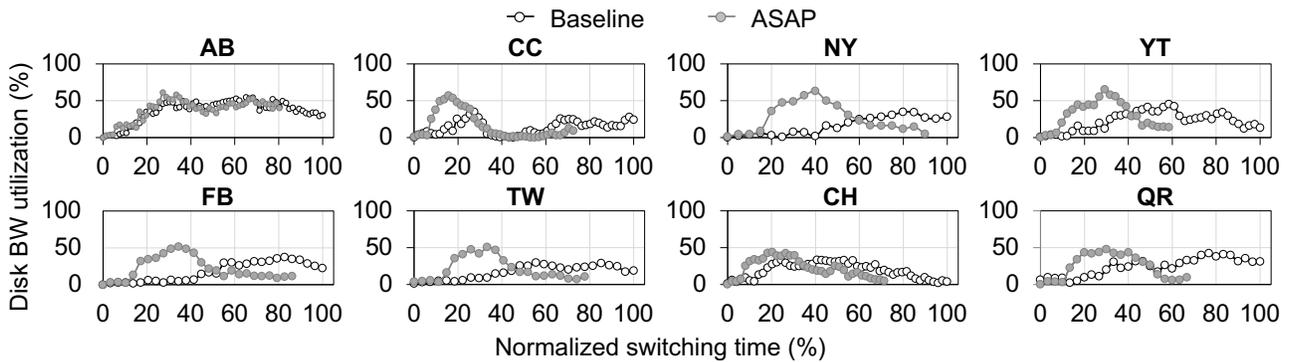


Figure 14: Disk bandwidth utilization. X-axis is a timeline normalized to baseline’s switch time.

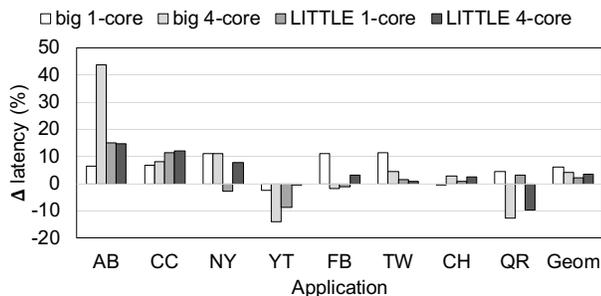


Figure 15: Switching latency changes depending on different core scheduling policy compared to ASAP’s core scheduling policies. Positive latency change means that the static policy is worse than ASAP’s policy.

the application switch latency.

6.5 Efficiency of Core Scheduling

To quantify the effect of core scheduling of the file prepping threads (Section 5.2), we compare our policy on Pixel 4 with four other static policies: big 1-core, big 4-core, LITTLE 1-core, and LITTLE 4-core. For example, in LITTLE 4-core, four file prepping threads are scheduled on the four LITTLE

cores, and the threads are assigned the SCHED_NORMAL priority. And we enabled only file prepping to reduce performance deviation. Figure 15 shows the delta of application switch latency (the latency of the four naive policies minus the latency of our scheme). Each naive policy shows 1.06×, 1.05×, 1.02×, 1.04× times slower than ours on average. Hence, our performance advantage comes from the fact that our policy is versatile to different situations. For example, the big 4-core policy showed 14% and 13% better performance than our policy on YT and QR, however its performance falls dramatically on AB since AB utilizes both CPU and disk bandwidth intensively, so file prepping threads contended a lot with AB’s application threads. On the other hand, the LITTLE 4-core policy is better than ours in QR and AB, but it is vulnerable to applications requiring heavy file I/O because of the slow prepping speed.

6.6 Overhead

Anonymous SFE maintains an *Online Candidate Table*, *Prepping Target Table*, and anonymous fault buffer. Their peak size for 8 applications is 1MB, 2.5MB, and 0.5MB, respectively. The size of the *Offline Candidate Table*, *Prepping Target Table* and file fault buffer used by File-backed SFE is 1.5MB, 0.2MB, and 0.5MB, respectively, at their peak respec-

tively. On average ASAP uses about 800KB per application.

Access bit logging (clearing access bits at the beginning and inspecting them at the end of the switch time) extends the switch time by up to 14ms. Also, prepagating target management operations which opportunistically runs between the switch events takes 40ms CPU time in the worst case.

Finally, mis-prediction events result in extra fetch overhead, which could increase the energy consumption. On average, ASAP fetches an extra 10MB for anonymous pages and file-backed pages, respectively. Also the peak throughput of decompression and the disk bandwidth are 2GB/s and 600MB/s on Pixel 4, respectively. Therefore, each extra fetch takes tens of milliseconds. When the peak power of UFS 2.1 [33] and TDP of Snapdragon 855 [32] are considered, these extra fetches require negligible overhead. Actually, we expect ASAP to save the energy consumption of the entire device including other components (e.g., display) because ASAP reduces the total switch latency. Thus, this marginal energy overhead can be easily offset.

7 Related Work

Efficient Memory Management in Mobile Systems. Modern mobile systems reclaim free pages by killing the least essential applications (e.g., low memory killer in Android [26]). The traditional low memory killer selects a victim process by considering the priority and the number of pages of application only. SmartLMK [19] proposes to kill an application to minimize the expected user-perceived application performance by carefully considering application usage statistics and application launch times. However, killing an application process is the most aggressive policy in memory reclamation [6], and whenever a killed application is launched again, it takes a large amount of computation and I/O operations, which can increase the user-perceived launch latency and the energy consumption of mobile devices [21, 24]

To end this senseless killing, Marvin [21] swaps out predicted unlikely-to-be-used pages to disks using ahead-of-time swap by modifying Android runtime (ART). Similarly, SmartSwap [35] includes process-level early page swap based on the prediction result but by addressing kernel codes. A2S [18] combines the low memory killer and the compressed swap together by carefully selecting the victim pages for swap-out and the victim process to kill. Acclaim [25] prioritizes pages of foreground processes over those of background processes during swapping. Kwon et al. [20] propose to swap-out GPU buffers of background processes to relieve memory pressure on mobile devices. Chae et al. [8] propose to extend the swap space of mobile systems to the cloud.

Accelerating Application Launch. Numerous studies have been conducted to shorten the application launch time, and most have tried to prefetch data effectively [12, 15, 28, 30, 34]. FAST [15] profiles I/O sequences during application launches and uses the profiled sequences for data prefetching.

FALCON [34] adopts machine learning to predict the users' application usage pattern. It predicts the next application a user is going to use and preloads the contents of the predicted applications. Nagarajan et al. [28] uses collaborative filtering to predict the impending applications while PREPP [30] uses prediction by the partial matching technique. IORap [12] in Android 11 profiles the required I/O during several cold-runs of an application and predicts which I/O will be required and does it in advance. These works only focus on predicting applications or I/O patterns during application launch events. However, our work predicts I/O patterns or memory access footprint during application switch events.

Efficiently Utilizing Disk I/O Bandwidth. The disk I/O performance is important to the user-perceived application performance. Accordingly, the efficient use of disk I/O is important. SmartIO [29] discovers that read I/O operations are penalized by write I/O operations and proposes to prioritize read I/O operations over write ones. Joo et al. [14] finds that swap I/O patterns for page faults are not efficient due to their small and random I/O request patterns. To overcome these inefficiencies, they insert pads to build large sequential I/O requests, which is more efficient in flash-based disks. FastTrack [11] prioritizes I/O requests from foreground applications over those from background ones throughout the entire I/O stack. These approaches are complementary to our work in terms of improving the I/O efficiency during disk access.

8 Conclusion

The goal of this paper is to improve user experience on mobile devices, focusing on the application switch, which is one of the most important user interactions. We proposed (ASAP), an adaptive prepagating scheme that accurately retrieves pages ahead of time that are expected to be accessed during application switch by fully exploiting the available system resources. Our experimental results based on real-world Android OS applications show that ASAP can reduce the application switch latency under memory pressure by 22.2% and 28.3% on representative high-end and mid-end smartphones, respectively. While ASAP was evaluated in the context of the application switch, we believe that it can easily be extended to reducing application launch time as well.

We open sourced Linux kernel we modified for ASAP. The code is available at <https://github.com/SNU-ARC/atc21-asap-kernel>.

Acknowledgments

We thank Lin Zhong for shepherding this paper. This work was supported by a research grant from Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1702-52. Hongil Yoon and Jae W. Lee are the corresponding authors.

References

- [1] Activity manager. <https://developer.android.com/reference/android/app/ActivityManager>.
- [2] Android Debug Bridge. <https://developer.android.com/studio/command-line/adb>.
- [3] Memory allocation among processes. <https://developer.android.com/topic/performance/memory-management>.
- [4] The Average Size of the U.S. App Store's Top Games Has Grown 76% in Five Years . <https://sensortower.com/blog/ios-game-size-growth-2020>.
- [5] App Startup Time. <https://developer.android.com/topic/performance/vitals/launch-time>.
- [6] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.
- [7] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '91*, page 181–186, New York, NY, USA, 1991. Association for Computing Machinery.
- [8] D. Chae, J. Kim, Y. Kim, J. Kim, K. Chang, S. Suh, and H. Lee. CloudSwap: A cloud-assisted swap mechanism for mobile devices. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 462–472, 2016.
- [9] Tao Deng, Shaheen Kanthawala, Jingbo Meng, Wei Peng, Anastasia Kononova, Qi Hao, Qin Hao Zhang, and Prabu David. Measuring smartphone usage and task switching with log tracking and self-reports. *Mobile Media & Communication*, 7:205015791876149, 04 2018.
- [10] fio - flexible I/O tester rev.325. https://fio.readthedocs.io/en/latest/fio_doc.html.
- [11] Sangwook Shane Hahn, Sungjin Lee, Inhyuk Yee, Donguk Ryu, and Jihong Kim. FastTrack: Foreground app-aware I/O management for improving user experience of android smartphones. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 15–28, Boston, MA, July 2018. USENIX Association.
- [12] Android I/O read ahead process. <https://medium.com/androiddevelopers/improving-app-startup-with-i-o-prefetching-62fbb9c9020>.
- [13] iOS Memory Deep Dive. <https://developer.apple.com/videos/play/wwdc2018/416/>.
- [14] Y. Joo, D. Seo, D. Shin, and S. Lim. Enlarging I/O size for faster loading of mobile applications. *IEEE Embedded Systems Letters*, 12(2):50–53, 2020.
- [15] Yongsoo Joo, Junhee Ryu, Sangsoo Park, and Kang G. Shin. Fast: Quick application launch on solid-state drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies, FAST'11*, page 19, USA, 2011. USENIX Association.
- [16] Conor Kelton, Jihoon Ryoo, Aruna Balasubramanian, and Samir R. Das. Improving user perceived page load times using gaze. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 545–559, Boston, MA, March 2017. USENIX Association.
- [17] CFS Scheduler. <https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html#scheduling-policies>.
- [18] Sang-Hoon Kim, Jinkyu Jeong, and Jin-Soo Kim. Application-aware swapping for mobile systems. *ACM Trans. Embed. Comput. Syst.*, 16(5s), September 2017.
- [19] Sang-Hoon Kim, Jinkyu Jeong, Jin-Soo Kim, and Seungryoul Maeng. SmartLMK: A memory reclamation scheme for improving user-perceived app launch time. *ACM Trans. Embed. Comput. Syst.*, 15(3), May 2016.
- [20] S. Kwon, S. Kim, J. Kim, and J. Jeong. Managing GPU buffers for caching more apps in mobile systems. In *Proceedings of the 2015 International Conference on Embedded Software (EMSOFT)*, pages 207–216, 2015.
- [21] Niel Lebeck, Arvind Krishnamurthy, Henry M. Levy, and Irene Zhang. End the senseless killing: Improving memory management for mobile operating systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 873–887. USENIX Association, July 2020.
- [22] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, Santa Clara, CA, February 2015. USENIX Association.
- [23] Gyun 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 *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 603–616, Renton, WA, July 2019. USENIX Association.
- [24] Joohyun Lee, Kyunghan Lee, Euijin Jeong, Jaemin Jo, and Ness B. Shroff. Context-aware application scheduling in mobile systems: What will users do and not do

- next? In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '16, page 1235–1246, New York, NY, USA, 2016. Association for Computing Machinery.
- [25] Yu Liang, Jinheng Li, Rachata Ausavarungnirun, Riwei Pan, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. Acclaim: Adaptive memory reclaim to improve user experience in android systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 897–910. USENIX Association, July 2020.
- [26] Low Memory Killer Daemon. <https://source.android.com/devices/tech/perf/lmkd>.
- [27] Why Brands Are Fighting Over Milliseconds. <https://www.forbes.com/sites/steveolenski/2016/11/10/why-brands-are-fighting-over-milliseconds/?sh=4f52e2f14ad3>.
- [28] Nagarajan Natarajan, Donghyuk Shin, and Inderjit S. Dhillon. Which app will you use next? collaborative filtering with interactional context. In *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, page 201–208, New York, NY, USA, 2013. Association for Computing Machinery.
- [29] David T. Nguyen, Gang Zhou, Guoliang Xing, Xin Qi, Zijiang Hao, Ge Peng, and Qing Yang. Reducing smartphone application delay through read/write isolation. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, page 287–300, New York, NY, USA, 2015. Association for Computing Machinery.
- [30] Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M. Marlin. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13, page 275–284, New York, NY, USA, 2013. Association for Computing Machinery.
- [31] V. J. Reddi, H. Yoon, and A. Knies. Two billion devices and counting. *IEEE Micro*, 38(1):6–21, 2018.
- [32] Qualcomm Snapdragon 855. <https://www.notebookcheck.net/Qualcomm-Snapdragon-855-SoC-Benchmarks-and-Specs.375436.0.html>.
- [33] High Performance Universal Flash Storage (UFS) Solutions. https://www.samsung.com/semiconductor/global.semi.static/White_Paper_Samsung_UFS_Card_1806.pdf.
- [34] 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.
- [35] X. Zhu, D. Liu, K. Zhong, Jinting Ren, and T. Li. Smartswap: High-performance and user experience friendly swapping in mobile systems. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2017.

PYLIVE: On-the-Fly Code Change for Python-based Online Services

Haochen Huang*, Chengcheng Xiang*, Li Zhong, Yuanyuan Zhou
University of California, San Diego

Abstract

Python is becoming a popular language for building online web services in many companies. To improve online service robustness, this paper presents a new framework, called PYLIVE, to enable on-the-fly code change. PYLIVE leverages the unique language features of Python, meta-object protocol and dynamic typing, to support dynamic logging, profiling and bug-fixing without restarting online services. PYLIVE requires no modification to the underlying runtime systems (i.e., Python interpreters), making it easy to be adopted by online services with little portability concern.

We evaluated PYLIVE with seven Python-based web applications that are widely used for online services. From these applications, we collected 20 existing real-world cases, including bugs, performance issues and patches for evaluation. PYLIVE can help resolve all the cases by providing dynamic logging, profiling and patching with little overhead. Additionally, PYLIVE also helped diagnose two new performance issues in two widely-used open-source applications.

1 Introduction

Python has gained wide adoption on developing online services. Many companies use Python to build their main online platforms. For example, Google's Youtube front-end server is built using Python [37, 51, 96]. Instagram's web services and Quora's main web services are also built with Python [36, 81, 94]. In addition to commercial companies, many open-source projects also use Python to build various frameworks for online services. Table 1 shows six categories of them, including web frameworks, e-commerce and message queues, etc.

These online services have critical requirements on high availability. A recent survey shows that 99.99% of uptime (i.e., 52.6 minutes per year) has become the minimum availability standard for most services [57]. A report from Statista shows that in 2020 one hour server downtime can cause more than \$301K cost for 88% of companies and more than \$5 million cost for 17% of companies [87].

However, high availability becomes challenging when there are code changes to apply to systems of running services. Such changes include adding logs to gain more diagnostic information, instrumenting programs to profile performance bottlenecks and applying patches to fix bugs.

*Co-first authors.

Category	Server Frameworks (Github stars)
E-commerce	Odoo (17.5k), Saleor (7.8k), Oscar (4.3k)
Web framework	Django (45k), Flask (48k), Pyramid (3.3k)
Web Server	Gunicorn (6.8k), Tornado (19.1k)
Message queue	Celery (14k), huey (2.7k), rq (6.6k)
Cache backend	Django-cacheops (1.1k), Beaker (429)
FTP server	pyftplib (1k), fbtftp (325), pyrexecd (202)

Table 1: Popular Python-based frameworks for online services.

Motivated by Python's popularity and the demand of applying code changes while keeping high availability, this paper presents a new idea that leverages the unique language features of Python to perform dynamic code changes. Specifically, we design and implement a framework, called PYLIVE, that enables dynamically changing Python programs for on-the-fly logging, profiling, and bug-fixing on *production systems* without restarting them.

PYLIVE's capability to change code during production runs can be used by online services for various purposes including:

- (1) On-the-fly logging for diagnosing production-run errors.** When an online service exhibits some abnormal behaviors, engineers can use PYLIVE to dynamically add logs at certain locations to collect debugging information from production-run. The logs can be enabled only during certain time (e.g. when the load is light) to minimize the performance impact. Production-run information is useful for diagnosing challenging issues (e.g., resource leaking) that are hard to reproduce in a testing environment and only manifest after a long-time (e.g., weeks) running.
- (2) On-the-fly profiling for diagnosing production-run performance issues.** When an online service has performance issues observed for certain types of requests, engineers can use PYLIVE to dynamically profile a set of functions for a short period of time to collect production-run timing information to troubleshoot the issues. The capability to (1) dynamically start and stop profiling during production runs and (2) only profile a small set of specified functions allows engineers to troubleshoot elusive performance issues without introducing much performance overhead. These performance issues may only emerge in production-run but are hard to reproduce during offline testing, making it necessary to perform in-production profiling.
- (3) Urgent dynamic patching (bug-fixing or security**

patch). PYLIVE allows dynamically applying urgent patches to fix some critical bugs or security issues without stopping and restarting an online service. These bugs and issues can either cause major failures or open up vulnerabilities to attackers and thus needs to be patched as soon as possible.

PYLIVE complements the commonly-used system update practice—rollout deployment [7, 42, 84]. Rollout is not the best choice for dynamic logging and profiling for two reasons. First, rollout still requires a restart of each service instance, which can clear the key program states for diagnosis. These states (e.g., resource leaks) may only be reproduced after a long production run, which is undesirable to be cleared by rollout. Second, a rollout deployment is heavyweight and an overkill for just collecting logging/profiling information. For instance, sometimes only a few servers of a fleet exhibit abnormal behaviors because of their unique memory states. If engineers want to diagnose the issue by rolling out a patch with new logging statements, this patch needs to be batched together with many other patches and will not be applied until the next deployment (wait for a few hours or even a few days). PYLIVE provides a better solution from both perspectives. It requires no restart of service instances so it retains the issue states for logging/profiling. In addition, it enables dynamically adding logging/profiling statements to a running program quickly and also removes the statements flexibly.

PYLIVE is designed based on the unique language features of Python. Python is an interpreted language that supports the meta-object protocol [22, 58] and dynamic typing. The meta-object protocol enables programs to dynamically modify their own metadata, including function bodies/interfaces and class attributes, while dynamic typing allows changing variable types during running. This makes dynamic code change much easier for Python than for compiled languages (e.g., C/C++) and other interpreted languages that do not support the full meta-object protocol or dynamic typing (e.g., Java):

- For compiled languages, dynamically changing a program requires many complex transformations to its code (e.g., patch functions) and memory layout (e.g., load new code into memory), as shown in previous work [38, 46, 54, 55, 68, 76, 77]. At run time, a compiled program’s code is binary code and its memory layout is fixed in different segments. Modifying binary code or memory layout may introduce safety concerns.
- Some other interpreted languages, like Java, do not support the full meta-object protocol or dynamic typing. For example, Java does not support many types of dynamic changes, like adding/deleting methods or changing methods’ signatures. Java is also statically-typed, making it hard to change variable types. For such languages, implementing dynamic code change requires modification to the language runtime (e.g., JVM), as in previous work [71, 78, 90]. This introduces portability concerns as different versions of runtime may be used by different systems.

PYLIVE makes two main contributions: (1) PYLIVE realizes safe and portable dynamic code change by leveraging Python’s language features. PYLIVE is safe as it requires no low-level transformations of machine code or memory layout. PYLIVE is portable as it relies only on the interfaces provided by standard Python interpreters and thus can be easily adopted by existing Python-based systems. (2) Besides patching, PYLIVE also enables instrumentation-based code changes for dynamic logging and profiling. PYLIVE provides convenient interfaces to flexibly instrument customized code to a selected set of functions at running time. This is useful for collecting diagnostic information in production-run systems without causing significant performance degradation.

We evaluate PYLIVE with 20 *real-world* cases from seven widely-used Python-based applications (including the popular Django framework and Gunicorn used by Instagram [35]). All these applications have been deployed in various companies to serve millions of customers [5, 13, 23, 26, 27, 34]. PYLIVE successfully helps resolve the cases with on-the-fly logging, profiling and patching with little overhead. Additionally, PYLIVE has helped two widely-used open source e-commerce applications diagnose two new performance issues. We also measure the performance benefit of PYLIVE by comparing it with restarting services to apply code changes. During normal time (no code change), PYLIVE’s overhead is negligible. Upon a code change, PYLIVE causes no downtime and has little performance degradation (<0.1%). In comparison, restarting the applications to apply changes can cause 2-17 seconds downtime and take up to 4.5 minutes to warm up (up to 90% performance downgrade during warmup).

2 Background

This section briefly describes the unique language features of Python that enables PYLIVE’s dynamic code change.

Meta-object protocol. Python is designed with a full support of the meta-object protocol [22, 58]. A meta-object is an object that contains a program’s metadata, including types, interfaces, classes and methods, etc. The meta-object protocol provides programming interfaces for programs to manipulate these metadata at runtime. For example, a new method named `A` can be dynamically added to class `C` simply with `C.A = D` (`D` is `A`’s definition). Similarly, an existing function’s code body can also be changed at runtime with `A.__code__ = D.__code__`. Once it is changed, the new `D` is called when `A` is invoked. Other supported changes include changing a function’s interface, adding a new field to a class, changing a variable’s type, etc. All the above changes are supported by the standard Python interpreter [32].

Dynamic typing. Python defines and checks variable types at running time, which makes it easy to dynamically change them. For instance, a Python variable `a` can be changed from

a string to a bool. And at running time each time before a is used, a type checking is performed. A wrong typed call `compare(a,b)` is detected by the Python interpreter, which will throw a runtime exception. In comparison, for Java, it is impossible to dynamically change a variable's type without changing the underlying JVM. For C/C++, this is also almost impossible as a string is passed by pointer while a bool is passed by value.

Python bytecode. A Python interpreter stores and interprets programs in bytecode, providing an opportunity for dynamically instrumenting code. Compared to machine code, bytecode is much easier to analyze and change with automatic tools. First, it is architecture-independent. Although Python can run on X86-64, ARMv7 and other architecture, it has the same bytecodes for all architectures. So the same tool can be used for Python on all platforms. Second, bytecode also has fewer instructions than machine instructions. Python 3.8 bytecode only has 112 instructions, while X86 alone has 1503 instructions, let alone all various architectures. Third, Python bytecode retains more type information than machine code.

3 PYLIVE Framework

PYLIVE is a runtime framework that accepts dynamic change requests from engineers and applies them dynamically into production-run systems without a restart. PYLIVE can be used for on-the-fly logging, profiling and bug-fixing.

In this section, we begin with the design objectives (§3.1) and interfaces (§3.2) of PYLIVE. We then discuss the three challenges faced by PYLIVE: (1) How to support dynamic changes for function interface, function body and data structure? (§3.3) (2) How to identify safe change points to apply a change without causing inconsistency problems? (§3.4) (3) How to update programs with multi-threads and multi-processes? (§3.5)

3.1 Design Objectives

PYLIVE is designed with the following objectives:

(1) *General.* PYLIVE's generality comes from three aspects: (a) it requires no change to the standard Python interpreter. Therefore, engineers need not to download a modified interpreter, which may not be compatible with their systems. (b) PYLIVE is also general on the types of changes supported. It supports not only changes to function body, but also changes to function interfaces (e.g. add one more parameter) and data structures (e.g. add one more field). (c) Since most online services have multiple threads/processes, PYLIVE also provides support for multi-threads and multi-processes.

(2) *Flexible.* PYLIVE is flexible from two perspectives. First, PYLIVE is flexible in terms of *when* to apply a change and *when* to revert a change, all based on engineers' requirements.

This can help engineers collect logs for a short amount of time to minimize the performance impact. For example, they can perform on-the-fly logging or profiling only during light-load time. Second, PYLIVE is also flexible with *where* to profile or log. PYLIVE allows engineers to specify which modules or functions to instrument logging/profiling code.

(3) *Consistent and Safe.* Dynamic changes to a running program need to be performed at a carefully selected execution point (aka, a safe point) to avoid inconsistency problems. For instance, changing an `unlock` function to its new implementation after an old `lock` function is already executed may cause inconsistency, leading to incorrect states. Unfortunately, choosing a safe point for general changes has proved to be undecidable [53]. So PYLIVE relies on engineers' knowledge to decide when a change is safe to happen: either when the changed functions are not executing, or a user-specified check function (e.g., specifying a lock is not held) returns true.

(4) *Low Overhead.* PYLIVE is designed to impose as little overhead as possible. At normal time when no change needs to be applied, the PYLIVE thread is sleeping and simply waiting for engineers' inputs. Once engineers instruct it to make code changes, PYLIVE's thread is woken up to perform the change. Once a change is already applied in this target program's metadata, PYLIVE gets to sleep again and is no longer involved in the execution of the target program.

(5) *Little Human effort.* PYLIVE aims to minimize engineers' efforts in using it. PYLIVE itself is downloaded as a small Python library that can be easily installed. Only two lines of Python code are needed to set up PYLIVE at the initialization of the target program. To insert a dynamic change, PYLIVE only needs engineers to write a small Python snippet to specify what needs to be changed. As shown in our evaluation of 20 real-world cases from seven widely-used Python software systems (cf. Table 3), each change specification needs only 7-13 lines of code).

3.2 PYLIVE's Interfaces

To make it easy to use, PYLIVE allows engineers to write the specification for dynamic code change in Python code. To enable dynamic changes for various purposes, PYLIVE supports two change interfaces: `instrument` and `redefine`.

Instrument. The `instrument` interface can instrument code to specified locations in certain functions or modules. It is useful for instrumenting log statements or profiling code to diagnose bugs or performance issues. The interface is:

```
instrument(scope, jointpoint_callback, time).
```

`scope` is a list of function/module names that need to instrument. When only the module name is given, all functions in it are instrumented. `jointpoint_callback` is key-value dictionary of jointpoints (instrument location) and callback

code to instrument. PYLIVE supports different granularity of jointpoints: coarse-grained, such as function begin/end, and fine-grained, such as before/after a line and before/after a variable's definition. This allows engineers to flexibly customize the instrument locations. `time` allows engineers to specify when to perform an instrumentation and when to revert the instrumentation. `time` can be either a specific time or a function that decides if an instrumentation should be performed. Engineers may want to profile an online service only when it is lightly-loaded and only for a period of time.

Redefine. The `redefine` interface is for code changes that replace the definitions of existing functions and classes (data structures) with new ones. To perform such code changes, engineers use the following Python interface:

```
redefine(preFunc, old_new_map, safepoint).
```

`preFunc` is a user-defined Python function that engineers need to provide to execute before making the specified change. Inside `preFunc`, engineers can import new modules and perform various initialization tasks. `old_new_map` is a key-value dictionary that specifies the changes. Each pair

```
{'old_func/old_class': new_func/new_class}
```

specifies an old function or class needs to be replaced by a new function or class. Engineers also need to provide the new function or class definition. To add a field to a class, just specify the field name and its initialization code with:

```
{'class.new_field': field_init}.
```

`safepoint` defines at what execution point it is safe to apply the specified change. It can be either "FUNC QUIESCENCE" or a user-specified consistency check function (cf. §3.4).

3.3 Support Dynamic Changes

Change function interface and body. PYLIVE supports changes on both function interfaces and code bodies. Changing function interfaces includes altering the number of parameters, parameter types, and function names. To make these changes, PYLIVE utilizes Python's meta-object protocol and dynamic typing. For parameter number changes, Python functions' parameters are defined in a list (i.e. `co_varnames`), and PYLIVE directly edits the list to add or remove parameters. For parameter type changes, Python uses dynamic typing, and so PYLIVE needs not explicitly change anything. For function name changes, PYLIVE defines a new function and modifies the callers to call the new function.

For function body changes, PYLIVE supports two types of changes: redefine a function body with a new one and instrument the old function body with extra statements (e.g., for logging or profiling). For both types, PYLIVE replaces functions' code object (`__code__`) as a whole, as code objects are immutable and can only be replaced by reference change.

For instrument changes, PYLIVE first copies the function's code object, builds an instrumented version by modifying its bytecode [10], and then sets the function's `__code__` to point to the instrumented code object.

PYLIVE may also need to change caller functions when changing the callee functions. For changes that modify callees' interfaces, PYLIVE needs to change all the callers' function body to call the new interface. PYLIVE expects that engineers include all changes to callers in the same patch as normal patching practice. For changes that only modify callees' function bodies, PYLIVE needs not to change the callers. This is because Python function calls are made by function names instead of addresses. Every time a function call happens, Python interpreters translate its name into address by looking up its metadata. Therefore, as long as the function metadata is updated (e.g. modify the `__code__` as discussed before), function calls can always be directed to the newest code objects. Note this differs from dynamic code changes in C/C++—they may need to update callers' function body as the function calls are made by address directly.

Change data structure. Data structure changes include changes to class attributes, object attributes and methods.

Class attributes are data fields defined in classes and shared by all the object instances. PYLIVE changes class attributes by modifying the namespace tables of the target classes. In order to hook class attributes access for profiling or debugging, PYLIVE adds `getter` and `setter` functions for attributes need to be changed. In Python, `getter` and `setter` are automatically called if an attribute is annotated as `property`.

Object attributes are more difficult to change since they are individually stored in different objects even though they are instantiated from the same class. In order to change an object attribute, it is necessary to go through all objects of the class and change each individually. Previous works typically need to refactor a system ahead of time so they can have Factory objects to keep track of all live objects at runtime [41]. PYLIVE utilizes Python's garbage collector (GC) to track live objects and modify each one when a change is requested. Specifically, PYLIVE calls `gc.get_objects()` to obtain a list of all live objects tracked by GC [14]. As Python uses reference counting to decide objects' liveness, this does not trigger a heap walk but returns a list immediately instead.

Methods are just functions defined in classes and so can be changed in the same way as global functions as described above. Methods' code is only stored in their classes instead of all instantiated objects, and so simply updating the classes' methods is sufficient to apply a change.

3.4 Identify Safe Change Point

Changing code at run time is not always safe. For instance, changing a function when it is executing may cause inconsistency problems. Therefore, dynamic code change systems

```

30 def file_move_safe(old_file_name, new_file_name):
    ...
58 fd = os.open(new_file_name,...)
59
60 try:
61     locks.lock(fd, ...)
    ...
65     os.write(fd, ...)
    ...
66 finally:
67     locks.unlock(fd)
    ...

```

Unsafe points
for changing
lock() / unlock()

Figure 1: An example of unsafe change points for a patch from Django [11]. It is unsafe to change `lock()` or `unlock()` when the program is executing between line 61 and 67, as the change can cause that a new `unlock()` to be called against an old lock, which can lead to undefined behavior.

need to carefully choose a safe execution point to apply a change. Unfortunately, choosing a safe point for general changes has proved to be undecidable [53]. As a result, it is necessary to have engineers' knowledge to choose a safe change point. PYLIVE categorizes safe points into different types and lets engineers select one based on the changes they want to make. Note that choosing the safe update point is only necessary when applying patches. PYLIVE can always apply code changes for logging and profiling as they only add code but do not change the existing code. PYLIVE supports two kinds of safe points:

Quiescence of the changed functions. This requirement means a change is only applied when the changed functions are not under execution. This is also the update point used by many previous dynamic code change systems [38, 38, 39, 86, 91]. It ensures that no function is executed with a mixture of old and new code during changes. PYLIVE provides automatic support for this safe point. To specify it, engineers only need to specify `safepoint='FUNC_QUIESCENCE'`.

PYLIVE supports function quiescence for both changing one function and multiple functions. When changing one function, PYLIVE directly takes advantage of Python meta-object protocol to guarantee the quiescence. In Python, when a function's code is changed, the change only takes effect the next time it is called. When changing multiple functions, PYLIVE checks every thread's stack for any changed function. If any changed function is on a stack, PYLIVE defers the change, retries the checks later and applies the change when no changed function is on any stack.

Consistent state check. When the changed functions modify shared states between them, function quiescence may not be enough for safety. Consider an example shown in Figure 1, two functions `lock` and `unlock` need to be changed, and both of them modify the lock state. Applying the change when the program is executing between the calls to `lock` and `unlock` is not safe, even though the functions themselves are not executed. The new `unlock` may be called with an old lock state

```

def state_check_func():
    for fd in all_fds():
        if locks.check_lock(fd) != locks.UNLOCK:
            return False
    return True

```

Figure 2: An example of state check function for the patch in figure 1. It returns true when the lock is not currently held.

and the behavior is undefined.

To address this, PYLIVE allows engineers to provide a customized boolean function to decide when it is safe to apply a change. This is also noted as state quiescence in previous work [48]. Engineers can easily write such boolean functions in normal Python code. Figure 2 shows the state check function for changing `lock` and `unlock`. It checks if no lock is held before applying the change. PYLIVE periodically evaluates it and only applies the change when it returns true.

Note for most code changes, it does not require any customized consistency check. In our evaluation with 20 real-world cases from seven widely-used Python programs, only a few cases require a simple consistency check.

Guidance for engineers. We provide guidance to help engineers identify and specify safe change points for their needs:

First, if the changed functions have no side effects or negligible side effects on execution state, engineers can specify function quiescence as the safe change point. For example, if the changed functions modify no non-local variables, perform no database write and only write a few logs, it is safe to update them as long as they are not under execution.

Second, if the changed functions have some non-negligible side effects on execution states, engineers need to identify the states that the side effects of the old and new functions will not affect each other. Specifically, the variables V_{old} defined and propagated from old functions f_{old} will not be used by new functions f_{new} , and vice versa. To ensure this, the target consistent states are either no variable in V_{old} is defined or all of them are dead. An example of this is shown in Figure 2 that no lock is held at the point of change. Such states may not exist or may not be easy to express in state check functions, and in such cases it may be better to perform a restart than to use PYLIVE.

3.5 Support for Multi-threads and Multi-processes

Multi-threads. A server program may have multiple threads to serve different user requests. Different threads have different program counters while sharing the same code and global variables. Therefore, it is not straightforward to apply a change at a given safe change point for multiple threads.

To change multiple threads correctly, PYLIVE applies a given change synchronously. The synchronous change is ensured by Python global interpreter lock (GIL). At any execution point, only one thread can hold GIL and so can get

executed [15]. Therefore, when PYLIVE is actively applying a dynamic change, all other threads to be changed are blocked. When applying changes, PYLIVE also explicitly holds GIL lock to make sure no other threads can preempt it [17].

Based on the type of safe point, PYLIVE applies changes differently. If the safe point is function quiescence, PYLIVE either immediately applies the change when only one function needs to change or check program stacks to make sure no target function on stacks when multiple functions need to change. Applying one function change is simpler because Python’s meta-object protocol ensures the change to not take effect during its execution. If the safe point is a consistent state check, PYLIVE first executes the check function provided by engineers. If the consistent check succeeds, PYLIVE then applies the change. If it fails, PYLIVE sets a timer t and goes to sleep to let other threads execute. The timer will wake up PYLIVE later to perform a state check again. The timer t is configurable by engineers. After several attempts, if it still fails, PYLIVE will give up and report an error to engineers.

Multi-processes. Online services may use multiple processes, and dynamic code change needs to be applied to all of them. Different Python processes reside in different address spaces and share code through copy-on-write. When code is changed in a process, a copy-on-write happens and other processes will continue to use the old code. As such, dynamic code change needs to be performed explicitly in all processes. PYLIVE adopts a controller-stub architecture to communicate changes to all processes. A stub is a change thread residing in a target process. PYLIVE starts one stub thread for each target process at its starting time. A stub thread listens to a controller for patches and applies the received patches at a safe change point. A PYLIVE controller is a standalone process that accepts engineers’ change input and sends the specified code change to the stub thread in each process.

4 Use Cases

PYLIVE enables three types of use cases that require a running system to be dynamically changed.

4.1 On-the-fly Logging for diagnosis

Systems may exhibit abnormal behavior during running. To collect run time info for diagnosis, engineers may want to add new log messages dynamically without restarting services.

An example of this is the diagnosis of a bug [28] from the Shuup [24] e-commerce system. This bug is related to its shopping cart: when some users click “add to cart”, the product is not added to the cart. This prevents users from purchasing products and causes direct revenue loss to businesses. Since the bug has no error logs, it is quite challenging for engineers to diagnose it off-line.

Figure 3 shows how engineers can use PYLIVE to add log messages to diagnose the issue. Engineers direct PYLIVE

```
# callbacks to instrument
# logging right/left-hand variables in each line
def call_b(_righthands):
    logging.info(_righthands)
def call_a(_lefthands):
    logging.info(_lefthands)

# instrument code to every line in two functions
instrument(scope=['...add_product',
                 '..._find_product_line_data'],
          jointpoint_callback={line_before: call_b,
                              line_after: call_a},
          time='24:00-2:00')
```

Figure 3: PYLIVE’s dynamic logging spec for an urgent, real world bug in Shuup e-commerce system [28]. This spec tells PYLIVE to dynamically instrument code to log some variable values in two functions `add_product` and `_find_product_line_data` for a period of time. `line_before` and `line_after` are two joint-points PYLIVE provides to instrument code before and after each line in functions.

to add line-by-line logs in two functions `add_product` and `_find_product_line_data`. Engineers also specify to only collect logs during light-load time (24:00-2:00).

4.2 On-the-fly Profiling

Performance issues often occur in production as systems have more and more features and scale up to a larger size. When such an issue emerges, engineers may want to enable profiling to certain parts of a system during production run.

An example [21] of such issues is from the Oscar e-commerce system. This issue happens when there are a lot of product categories in Oscar. The issue causes a performance downgrade in many pages displayed to customers in Oscar, preventing customers from buying products.

Figure 4 shows how to use PYLIVE to dynamically instrument code to profile the system. Engineers instruct PYLIVE to instrument customized profiling code into the methods in two classes, `AbstractCategory` and `CatalogueView`, that are speculated to be related to the issue.

4.3 Dynamic Patching

Online services frequently have urgent bugs (e.g., security bugs) that need to be patched as quickly as possible to minimize damages since they may cause information leakage/system compromise and prevent customers using online services.

An example [1] of such patches is from Django. It fixes a severe cross-site scripting (XSS) [8] issue, CVE-2019-12308 [9]. The issue is scored as “6.1” since it can expose malicious URLs as clickable links to victim users and direct them to vulnerable sites. Django developers quickly post a security release [12] to fix the vulnerability and encourage all online services that use Django to apply it as soon as possible.

```

# profiling code to instrument
def call_b(start):
    start = time.time()
def call_a(start):
    logging.info(time.time()-start)

# instrument code to all functions of two classes
instrument(scope=['...AbstractCategory.*',
                '...CatalogueView.*'],
          jointpoint_callback={func_before: call_b,
                              func_end: call_a},
          time='24:00-2:00')

```

Figure 4: On-the-fly profiling using PYLIVE to diagnose a critical performance issue occurred in Oscar ecommerce system [21]. This example requires PYLIVE to instrument code to profile the execution of every method in two classes AbstractCategory and CatalogueView for a period of time.

Figure 5 shows part of the patch and the change spec that engineers need to provide for PYLIVE to dynamically apply it. This patch is non-trivial to be dynamically applied, as it changes both function interfaces and data structures. It adds a new parameter `validator_class` to the `__init__` function and adds a new attribute `self.validator` to `AdminURLFieldWidget`. The change spec calls the `redefine` interface with three arguments: `preupdate` specifies that PYLIVE needs to import a new class `URLValidator` before applying the change; `old_new_map` indicates that the original `__init__` will be replaced with the new code. `safepoint='FUNC QUIESCENCE'` tells PYLIVE to apply the change when the changed functions are quiescent. This requirement is safe enough in the case as there is no inter-dependency between the changed functions.

5 Evaluation

5.1 Methodology

We evaluate PYLIVE with 20 cases from seven Python-based real-world applications, as shown in Table 2. These applications are deployed in many companies, serving millions of customers [5, 13, 23, 34]. Django is a popular web framework that powered over 94,319 websites, of which many are for e-commerce [85]. Unicorn is a production web server used by many big companies for their main services, such as Instagram [35]. All the online services need to be almost non-stop since any downtime can result in revenue loss.

To evaluate PYLIVE’s benefit, we compare PYLIVE with a typical restart approach: modify code for logging/profiling/patching offline, stop the services and restart the services *immediately*. To precisely measure the restart impact, we only restart the Python part of a service, which does not restart other parts (e.g., database) to avoid the impact of warming up their cache. For profiling, we also compare PYLIVE with cProfile [47], which is Python’s official profiling tool for collecting comprehensive profiling information in test environments.

```

# patch: add a parameter validator_class
# add an object attribute validator
class AdminURLFieldWidget(...):
    def __init__(self, attrs=None,
                validator_class=URLValidator):
        self.validator = validator_class()
        ...

# change specs.
def preupdate_call():
    from django.core.validators import URLValidator

redefine(
    preupdate = preupdate_call,
    old_new_map={
        '...AdminURLFieldWidget.__init__': __init__,
        safepoint='FUNC QUIESCENCE'})

```

Figure 5: A real world security patch to Django [1] and PYLIVE’s dynamic change spec for it. This patch adds a parameter to function `__init__` and adds an object attribute validator to class `AdminURLFieldWidget`. Other part of the patch is omitted due to space limit. The change spec indicates: `preupdate` — import `URLValidator` before the change; `old_new_map` — replace `AdminURLFieldWidget.__init__` with a new one; `safepoint` — apply the change when the changed function is quiescent.

Applications	Category	Logging	Profiling	Patching
Django [33]	Web framework	1	0	2
Gunicorn [16]	Web server	0	0	1
Oscar [6]	E-commerce	1	2	1
Odoo [25]	E-commerce	1	1	2
Shuup [24]	E-commerce	1	0	1
Pretix [27]	E-commerce	1	0	1
Saleor [61]	E-commerce	1	1	2
Total		6	4	10

Table 2: 20 real-world cases evaluated in our experiments. They are from seven widely used Python-based server applications that have powered many commercial e-commerce and ad-based web services including Instagram, serving millions of customers.

Note PYLIVE is not a substitute for cProfile as it collects less information than cProfile. However, as we will present in the results, some cases only need little dynamic information to diagnose. We conduct this comparison to study the benefit that PYLIVE can bring for such cases.

Each application is set up on a machine with a 2.30GHz CPU (6 core), 16GB Memory and 256GB SSD. Each application runs with 2 processes and 4 threads/process. Each application is initialized with ~2000 web pages. To mimic real-world workloads, JMeter [4] is used to generate random web page accesses. The JMeter client is started with 8 threads and can generate up to 15K requests/second.

We use throughput as the performance metric and normalize it to the max throughput of normal service run (41-752 requests/second). All the experiments are conducted within a LAN, which ensures that network is not the bottleneck.

5.2 Overall Performance Results

Overall, PYLIVE avoids 2-17 seconds downtime and avoid up to 4.5 minutes warmup time, during which the performance downgrade can be 55%-90%. PYLIVE causes negligible (<0.1%) overhead during normal run as well as applying changes. PYLIVE causes 0.1%-1.4% overhead during profiling. Compared with cProfile, PYLIVE's selective instrumentation avoids 10.5%-33.6% overhead.

Figure 6,7 show the results of eight representative cases. Two newly identified performance issues and the other twelve existing real-world cases have similar results and due to space limit are put online [3].

For **logging cases**, PYLIVE's benefits mainly come from avoiding the time to restart and warm up. The service restart is relatively fast (2-17 seconds), but the warmup takes much longer time. Our experiments set up applications with only ~2000 web pages, but the warmup still takes 2.3-3 minutes.

For **profiling cases**, PYLIVE makes the performance impact caused by profiling affordable in production-run systems. The benefit comes from two aspects. First, PYLIVE allows engineers to perform customized profiling, so they need not profile applications in a whole as with cProfile. The customized profiling is not a substitute for comprehensive profiling with cProfile because it collects less information. However, it's sufficient to diagnose many cases that only needs limited timing information, as shown later in our case studies (cf. §5.3). Second, PYLIVE avoids restart and warmup time (up to 4.5 minutes), which is needed by cProfile. With PYLIVE, the performance downgrade during profiling is 0.1%-1.4%. While with cProfile, the performance downgrade can be 11%-39%.

For **patching cases**, PYLIVE can apply them dynamically with almost no performance downgrades. This benefits urgent security patches, for which waiting for the next rollout can be dangerous. Our evaluation includes 5 security patches and PYLIVE successfully applied them on-the-fly.

5.3 Case Studies

This section dives into the details of eight representative cases. The remaining twelve cases evaluated are similar and due to space limit we put them online [3].

Case 1: Diagnose a purchase bug in Shuup [28]. This case is about diagnosing a bug related to the shopping carts of Shuup [24], a widely-deployed e-commerce website. As mentioned in §4.1, the bug causes an error in production and prevents customers from adding new items to shopping carts. To help diagnose it, PYLIVE dynamically instruments logging statements on the running application. Figure 6a shows that PYLIVE avoids 3 seconds downtime and 2.3 minutes warmup time. It imposes only < 0.1% performance overhead.

Case 2: Diagnose a payment bug in Odoo [74]. Odoo [25] is an e-commerce website and this bug prevents customers from paying an order. It is an "urgent" bug as it results in

business loss. Odoo engineers diagnosed it by adding two logging statements and restarting the service. With PYLIVE, the logging statement can be added on-the-fly with 11 LOC. As shown in Figure 6b, PYLIVE avoids 4 seconds service downtime with < 0.1% overhead. Differing from other applications, Odoo does not have much cache and so requires little warmup time.

Case 3: Diagnose a purchase bug in Pretix [20]. Pretix [27] is an ticket-booking website that allows event organizers to sell event tickets online. In this case, when customers request a PayPal refund, it fails silently with no error messages. PYLIVE can dynamically instrument logging code to diagnose the reason. Figure 6c shows that PYLIVE successfully avoids 17 seconds downtime and 3 minutes warmup (by restarting Pretix) with < 0.1% performance overhead.

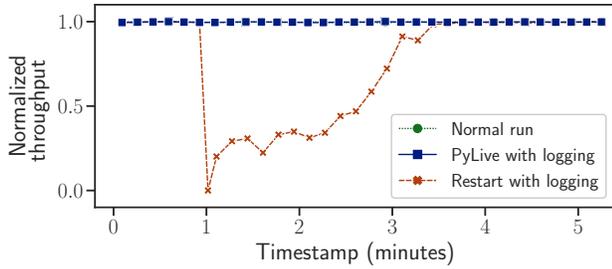
Case 4: Profile a main web page in Saleor [83]. This case is about diagnosing a slowly-loaded web page. This case is difficult to reproduce in testing as it only emerges when the product category grows to large. Currently, engineers use cProfile to profile the whole application [83]. Enabling cProfile needs an application restart, causing downtime and warmup time as shown in Figure 6d. Also, cProfile profiles every function, so after warmup it still imposes 35% performance downgrade.

PYLIVE can benefit the diagnosis in two ways. First, it can dynamically instrument profiling code into a running application. Figure 6d shows this can avoid 3 seconds downtime and 4.5 minutes warmup of Saleor services. Second, it can be customized to only profile the relevant functions suspected by engineers and thus reduces profiling overhead to only 1.4%.

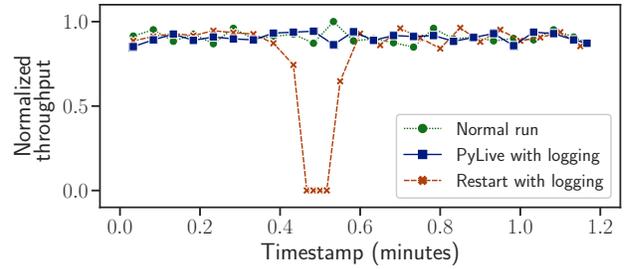
Case 5: Profile a slowly-loaded web page in Oscar [21]. This case is about diagnosing a slowly-loaded product-listing page. It happens when the number of products grows to large. PYLIVE enables dynamic profiling to Oscar with 9 LOC to specify the change. As shown in Figure 6e, PYLIVE causes only 0.5% performance overhead during profiling and nearly no overhead during normal run. In contrast, cProfile causes as much as 11% performance downgrades as well as 2 seconds of downtime and 3 minutes warmup time.

Case 6: Profile a slow action in Odoo [75]. This case is about diagnosing a slow receipt-validating action. It is hard to reproduce in testing as it only emerges when the database contains a large number of products and orders. PYLIVE enables dynamic profiling with 9 LOC. Figure 6f shows PYLIVE's performance benefit. PYLIVE causes only 0.1% performance overhead during profiling and nearly no overhead during normal run. In contrast, cProfile causes as much as 38.5% performance downgrades as well as 9 seconds of service downtime.

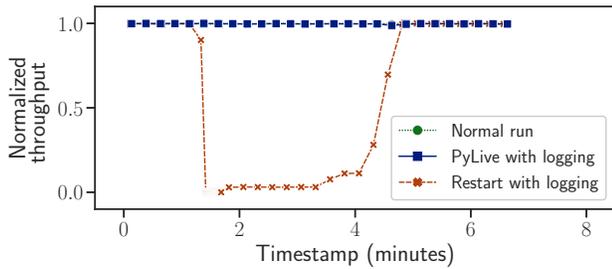
Case 7: Patch CVE-2019-12308 security vulnerability in Django [1]. This patch fixes a severe XSS security issue CVE-2019-12308 [9]. As we discussed in §4.3, it may lead users to click into malicious websites and can possibly affect



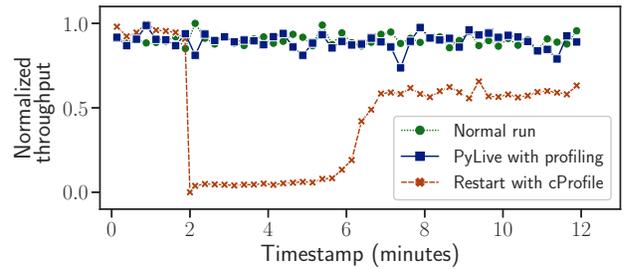
(a) **Shuup: on-the-fly logging to diagnose a payment bug [28].** PYLIVE causes < 0.1% overhead only when adding logs. In comparison, restarting causes 3 seconds of downtime and needs 2.3 minutes to warmup.



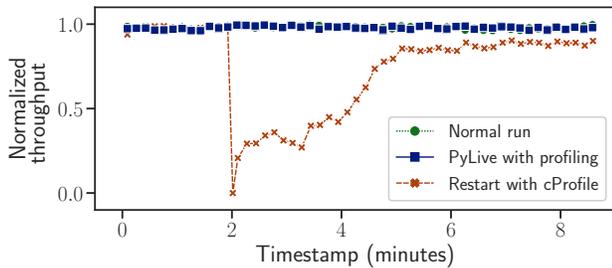
(b) **Odoos: on-the-fly logging to diagnose a shopping-cart bug [74].** PYLIVE causes < 0.1% overhead only when adding logs. In comparison, restarting causes 4 seconds of downtime. Odoos does not have much cache so has a short warmup time.



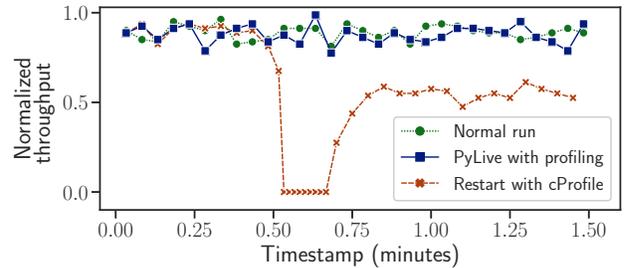
(c) **Pretix: on-the-fly logging to diagnose a payment bug [20].** PYLIVE causes < 0.1% overhead only when adding logs. In comparison, restarting causes 17 seconds of downtime and needs 3 minutes to warmup.



(d) **Saleor: on-the-fly profiling a long-loaded web page [83].** PYLIVE causes 1.4% overhead only during profiling. In comparison, restarting causes 3 seconds of downtime and needs 4.5 minutes to warmup. Using cProfile causes 35% overhead.



(e) **Oscar: on-the-fly profiling a long-loaded web page [21].** PYLIVE causes 0.5% overhead only during profiling. In comparison, restarting causes 2 seconds of downtime and needs 3 minutes to warmup. Using cProfile causes 11% overhead.



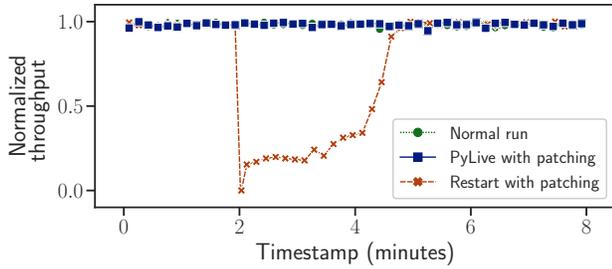
(f) **Odoos: on-the-fly profiling a long-loaded web page [75].** PYLIVE causes 0.1% overhead only during profiling. In comparison, restarting causes 9 seconds of downtime. Using cProfile to profile causes 38.5% overhead.

Figure 6: Throughput comparison of **three on-the-fly logging cases and three on-the-fly profiling cases** with PYLIVE in comparison with today’s practices—stop and restart with logging added and profiling enabled.

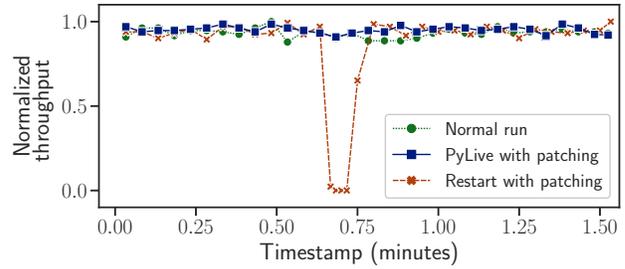
many users. This patch is non-trivial to be dynamically applied, as it involves adding a parameter to a method interface and adding a new class attribute [1]. With PYLIVE, this patch is allowed to be applied safely with 8 additional LOC. Figure 7a shows the performance benefit of PYLIVE and PYLIVE avoids 2 seconds of downtime and 2.8 minutes warmup time. PYLIVE dynamically applies the patch with < 0.1% overhead.

Case 8: Patch CVE-2018-1000164 in Gunicorn [52]. This patch fixes a HTTP Response Splitting Vulnerability [73]. It

has a severity score of “**7.5 High**” in the CVE system [72]. It can be exploited by various attacks, such as Cross-site Scripting (XSS), Cross-User Defacement, Hijacking [73]. The patch requires a modification to a function body. It can be dynamically applied with PYLIVE with only 13 additional LOC. As shown in Figure 7b, PYLIVE avoids 4 seconds of downtime, when a non-cached service runs on Gunicorn. PYLIVE introduces < 0.1% overhead while applying the patch.



(a) **Django: urgent security patching for CVE-2019-12308 [1].** Compared with restarting, PYLIVE avoids 2 seconds of downtime and 2.8 minutes warmup time, with $< 0.1\%$ performance overhead during patching.



(b) **Gunicorn: security patching for CVE-2018-1000164 [72].** Compared with restarting, PYLIVE avoids 4 seconds of service downtime, with negligible overhead during patching. Gunicorn’s workload is Odoo, which has little cache, so it takes a short time to warmup.

Figure 7: Throughput comparison of **two representative patching cases** with PYLIVE and restarting services.

Use Case	Software	LOC	Use Case	Software	LOC
Case1	Shuup	7	Case5	Oscar	9
Case2	Odoo	11	Case6	Odoo	9
Case3	Pretix	9	Case7	Django	8
Case4	Saleor	9	Case8	Gunicorn	13

Table 3: **Lines of code (LOC) of change specification for PYLIVE.** For patches, this only count extra code for PYLIVE.

5.4 Human Effort

PYLIVE requires only a little human effort to adopt it in real-world applications. To enable PYLIVE in a Python-based application, it only needs to add two lines of code in the application’s initialization stage. To apply dynamic change for different purposes, PYLIVE allows engineers to write Python code to specify the intended changes. Table 3 shows the lines of code (LOC) to specify the changes in the eight representative cases. For all cases, it requires only 7-13 lines of code to specify each change.

6 Limitations and Discussion

There are many kinds of code changes that PYLIVE cannot apply. First, PYLIVE cannot apply changes to long-running functions because dynamic changes only take effect next time when the functions are called. Fortunately, online services are usually request based and the major part is the request handling functions, which finish running in a short amount of time. Second, PYLIVE cannot apply patches that assume an initial program state. Patches for memory-leak bugs may need to reinitialize the program state to free the leaked memory, which needs a restart of the target program. Third, PYLIVE is not suitable for applying major changes to a target program. Such changes include adding new features, updating library versions, and refactoring the program structure. These changes may involve major changes to program states and

code logic of many functions. Therefore, it is hard for engineers to write code to initialize the states and to specify a safe update point that considers all the dependencies between the changed functions.

PYLIVE relies on engineers to specifies the safe update points for dynamic patching. PYLIVE targets on simple bug-fixing and security patches that only update a few functions and data structures. For these patches, the safe update points can be specified as when the targeted functions are not executing or when a customized state check passes (e.g. a lock is not held as in Figure 1). However, for more complex patches that change many interdependent functions and data structures, the safe update point may not be easy to specify. For such cases, it is safer to restart the target program than to use PYLIVE. Note the safe update point is only necessary for applying patches but not for logging or profiling. Code changes for profiling and logging can always be safely applied as they only add code but do not change the existing code.

PYLIVE cannot prevent errors introduced by buggy patches. PYLIVE expects that engineers thoroughly test their patches in a testing environment before dynamically applying them to production-run systems. For logging and profiling cases, PYLIVE wraps the instrumented code in try-catch blocks so that buggy logging or profiling code does not affect the normal program execution.

PYLIVE has two security implications. First, in terms of the type of code changes that can be made dynamically, PYLIVE does not expand the attack surface of Python’s own meta-object protocol. PYLIVE does not modify the Python interpreter to enable more types of code changes but just provides convenient interfaces purely based on Python’s meta-object protocol. Second, the introduction of a change controller (cf. §3.5) expands the attack surface from one single process to two processes. The change controller is an additional process that commands a target program process to apply a change dynamically. Therefore, it would be dangerous if attackers gain access to the change controller. It can be mitigated by

setting the change controller's permission to make it only executable by a privileged user. We also plan to implement PYLIVE's own access control for the change controller in future work.

PYLIVE's design and implementation are generally applicable to Python variants and other interpreted languages as long as they support three language features:

- Meta-object protocol—PYLIVE uses this to modify a program's code at running time (cf. §2);
- Dynamic typing—PYLIVE relies on this to modify variable types at running time (cf. §2);
- Interpreter interfaces to freeze non-current threads—PYLIVE uses them to pause the execution of any other threads to safely apply changes (cf. §3.5).

All three features are supported by popular python variants including Pypy [29] and Pyston [2], and so PYLIVE can be easily ported to them. PYLIVE can also potentially be ported to two other popular interpreted languages: JavaScript [19] and Ruby [30]. The first two features are directly supported by JavaScript and Ruby. The third feature can also be implemented in JavaScript and Ruby in different ways. JavaScript uses a single-threaded event loop model—at any time only one event handler is running and it cannot be preempted before its completion. Therefore, when PYLIVE is running in JavaScript, any other thread is ensured not running at the same time. Ruby's official interpreter YARV [31] has a similar GIL lock as Python's GIL, which allows PYLIVE to hold GIL in Ruby to prevent preemption as in Python (cf. §3.5).

7 Related Work

Dynamic Code Change for C/C++ and Java. Many works have been done for dynamic changing C/C++-based operating systems [39–41, 45, 49, 64, 86] and applications [38, 46, 54, 56, 63, 69, 82, 89]. While simple dynamic change (e.g. patch only function bodies) to OS kernels has been used in production, more general change to applications has not been widely adopted. General dynamic change usually requires many unsafe transformations to target programs including modifying both machine code and memory layout. This may introduce safety concerns in production. Contrarily, PYLIVE realizes general changes by utilizing Python's standard language features—meta-object protocol and dynamic typing, making it safer to be adopted in production.

Works on dynamic changing Java either need to modify JVM [71, 90] or rely on some unsafe operations of JVM [78], introducing portability and safety concerns in production. When running a Java program, JVM maintains many metadata such as method signatures and class attributes as internal data but provides no safe operations to modify them. However, it is necessary to modify these metadata in order to support general dynamic change. In comparison, PYLIVE makes use of Python's meta-object protocol to safely modify related metadata when dynamically changing Python programs. This

imposes no modification to a standard Python interpreter and so can be easily adopted in existing production systems.

Dynamic Code Change for Python. Pymoult [66] made a preliminary exploration on the feasibility of dynamically changing Python programs. As a preliminary proposal, it has no experimental result. More importantly, Pymoult relies on a special Python interpreter, Pypy, which is not fully compatible with the standard Python interpreter (i.e. CPython [32]). In order to use Pymoult, engineers need to port their systems to Pypy interpreter, which requires considerable human efforts. Contrarily, PYLIVE is based on the standard Python interpreter and so can be easily adopted in the field.

PyReload [93] is a dynamic code change tool based on the standard Python interpreter. However, it has two key limitations that prevent it to be practical. First, PyReload needs engineers to refactor a target program into modules, which requires huge human efforts. Second, PyReload only supports single-threaded programs. Considering that servers usually have multiple threads, PyReload is not suitable for server systems. In contrast, PYLIVE requires no refactor to the target program and supports updating multi-thread server programs.

Language level support for dynamic code change. Language level support for dynamic code change is not new. Besides Python, many other dynamically-typed programming languages, such as Common Lisp [88], Smalltalk [50], JavaScript [19] and Ruby [30], have provided support for meta-object protocol. Meta-object protocol can be directly used to update a single function/class; however, there are still many challenges in using meta-object protocol to practically update server programs, which usually needs to update multiple functions/classes and threads/processes. Very few works have been done on these challenges. Rivet [67] proposes interesting ideas to leverage JavaScript's reflection capabilities to debug single-threaded browser-side programs. But the ideas cannot be directly borrowed to update server programs, which usually have multiple threads/processes.

Focusing on Python, PYLIVE addresses three challenges of leveraging meta-object protocol to update server programs. First, to make it easy to update multiple functions and classes, PYLIVE provides two APIs: `Redefine` and `Instrument` (§3.2) and adopts the meta-object protocol and bytecode instrumentation to implement them (§3.3). Second, to make it safe to update multiple functions and classes, PYLIVE borrows ideas from previous works [38, 39, 48] and provides two different safe points (§3.4). Third, to support updating programs with multiple-threads and multiple-processes, PYLIVE proposes new synchronous update mechanisms based on Python GIL and a controller-stub architecture (§3.5).

Aspect-oriented programming. PYLIVE's `Instrument` interface is a type of aspect-oriented programming (AOP) [60]. AOP is a programming paradigm to break down independent program logic into different modules. A common usage of

AOP is to allow developers to write a function’s main logic and its logging code separately. The AOP framework then “weaves” the code together at compile time or load time. Several works also aim to enable dynamic weaving at run time for AOP [79, 80, 92]. PYLIVE’s `Instrument` interface is an AOP support for Python and is inspired by previous work on AOP interfaces for Java [59, 65]. However, AOP’s main target is to insert additional code without modifying the existing code. PYLIVE also provides a `Redefine` interface to modify the existing code of a running program, which is challenging especially when the target program has multiple threads/processes. To realize `Redefine`, PYLIVE further considered the challenge of supporting safe update points and multiple-threads/processes programs.

Dynamic Instrumentation. PYLIVE’s `Instrument` interface is related to previous works on dynamic binary instrumentation (DBI), including `Pin` [62], `Valgrind` [70], and `DynamoRIO` [43]. DBI enables modifying a running binary program at the machine instruction level and is usually used for logging and profiling a compiled program. However, DBI cannot be directly adopted for profiling a Python program in production. When DBI is used for a Python program, DBI is instrumenting the Python interpreter instead of the Python program. This can cause two folds of problems. First, the logging and profiling results are verbose and hard to understand by Python developers as they are mostly about the execution of the Python interpreter but not about the Python program. Second, this can introduce an unacceptable performance downgrade to the Python program as interpreting one line of Python code may need to run multiple lines of interpreter code. PYLIVE addresses these problems by instrumenting code at the granularity of Python bytecode. Therefore, the logging and profiling results can be directly mapped back to the Python program and so are easy to understand by Python developers. In addition, much less code is instrumented and so much less performance downgrade.

PYLIVE complements `DTrace` [44] on dynamic instrumentation. `DTrace` is a dynamic instrumentation framework for production systems. To enable `DTrace` for Python, it needs to embed “markers” in Python interpreters. This introduces additional compatibility requirements for Python interpreters to use `DTrace`. As noted by the official Python document [18], “`DTrace` scripts can stop working or work incorrectly without warning when changing CPython versions.” PYLIVE takes a complementary approach to instrument Python application code without modifying Python interpreters, avoiding the compatibility concerns.

Rollout Update. An alternative way to avoid whole system downtime is rollout update [7, 42, 84]. In rollout update, a cluster of servers are restarted one by one or batches by batches so that during an update there are still servers alive to serve users’ requests. However, for just collecting logging/profiling information, rolling out patches to a whole cluster at the

next deployment is heavyweight and an overkill. It would be handy to quickly apply a simple patch that temporarily logs extra information or collects extra metrics to some servers on-the-fly. Furthermore, rollout update is less effective for collecting diagnostic or profiling information for certain types of issues because rollout update still requires restarting every service instance. As a result, errors that appear only after a long running time, such as resource leaks and concurrency issues, may not reappear quickly after restarting to provide diagnostic information [95]. Finally, rollout update can still result in a subset of servers restarting and warming up before providing service at their full capacity. This means during the rollout update, the entire system will suffer from certain levels of throughput degradation.

8 Conclusions

In this paper, we proposed a framework called PYLIVE that leverages Python’s unique language features, meta-object protocol and dynamic typing, to support dynamic code change for on-the-fly logging, profiling and patching in production-run systems. PYLIVE only relies on standard Python interpreters and can be easily adopted by existing systems. We evaluated PYLIVE with seven widely-deployed Python-based systems for online services. PYLIVE successfully helped resolve 20 existing real-world cases from these systems with dynamically logging, profiling and patching. PYLIVE also helped two of the systems diagnose two *new* performance issues. In comparison to restart, PYLIVE avoids service downtime and warmup. PYLIVE imposes no overhead during normal run and negligible overhead during the change. For profiling, PYLIVE adds only 0.1%-1.4% overhead.

Acknowledgments

We greatly appreciate our shepherd, Larry Rudolph, and the anonymous reviewers for their insightful comments and feedback. We thank Stewart Grant, Rajdeep Das, Keegan Ryan, the Opera group as well as the Systems and Networking group at UCSD for helpful discussions and paper proofreading. This work is supported in part by NSF grants (CNS-1814388, CNS-1526966) and the Qualcomm Chair Endowment.

References

- [1] [2.2.x] Fixed CVE-2019-12308 – Made AdminURLFieldWidget validate URL before rendering clickable link. <https://github.com/django/django/commit/afddabf8428ddc89a332f7a78d0d21eaf2b5a673>.
- [2] A faster and highly-compatible implementation of the Python programming language. <https://github.com/pyston/pyston>.
- [3] Anonymous repo for PyLive evaluation result. <https://github.com/pyupdate/evaluation>.
- [4] Apache JMeter - Apache JMeter. <https://jmeter.apache.org/>.
- [5] Butterfly Network Case Study – Saleor Commerce. <https://saleor.io/case-study/butterfly-network/>.
- [6] Cases / Oscar - Domain-driven e-commerce for Django. <http://oscarcommerce.com/cases.html>.
- [7] Continuous Deployment at Instagram. <https://instagram-engineering.com/continuous-deployment-at-instagram-1e18548f01d1>.
- [8] Cross Site Scripting (XSS) Software Attack | OWASP Foundation. <https://owasp.org/www-community/attacks/xss/>.
- [9] CVE-2019-12308 Detail. <https://nvd.nist.gov/vuln/detail/CVE-2019-12308>.
- [10] Disassembler for Python bytecode. <https://docs.python.org/3/library/dis.html>.
- [11] Django patch: Changed the use of `fcntl.flock()` to `fcntl.lockf()`. <https://github.com/django/django/commit/195420259a5286cbeface8ef7d0570e5e8d651e0>.
- [12] Django security releases issued: 2.2.2, 2.1.9 and 1.11.21. <https://www.djangoproject.com/weblog/2019/jun/03/security-releases/>.
- [13] edX. <https://open.edx.org/blog/using-open-edx-ecommerce-module/>.
- [14] Garbage Collector interface. https://docs.python.org/3/library/gc.html#gc.get_objects.
- [15] GlobalInterpreterLock - Python Wiki. <https://wiki.python.org/moin/GlobalInterpreterLock>.
- [16] Unicorn - Python WSGI HTTP Server for UNIX. <https://unicorn.org/>.
- [17] Initialization, Finalization, and Threads. https://docs.python.org/3/c-api/init.html#c.PyGILState_Ensure.
- [18] Instrumenting CPython with DTrace and SystemTap. <https://docs.python.org/3/howto/instrumentation.html>.
- [19] Javascript Programming Language. <https://www.javascript.com/>.
- [20] Log the reason for failed PayPal refunds. <https://github.com/pretix/pretix/commit/5400d26c60b7a4fceb2c832419e63abfd785f0d>.
- [21] Long rendered page when a lot of categories products 1910. <https://github.com/django-oscar/django-oscar/issues/1910>.
- [22] Metaobject. <https://en.wikipedia.org/wiki/Metaobject>.
- [23] Multivendor Marketplace Platform - Enterprise Commerce Software. <https://www.shuup.com/>.
- [24] Multivendor Marketplace Platform - Enterprise Commerce Software. <https://www.shuup.com/>.
- [25] Odoo. <https://www.odoo.com/>.
- [26] Odoo Customer Reviews | Success Stories. <https://www.odoo.com/blog/customer-reviews-6>.
- [27] pretix – Reinventing ticket sales for conferences, festivals, exhibitions, ... <https://pretix.eu/about/en/>.
- [28] Product does not get added to basket if `force_new_line = True` #291. <https://github.com/shuup/shuup/issues/291>.
- [29] Pypy. <https://www.pypy.org/>.
- [30] Ruby Programming Language. <https://www.ruby-lang.org/en/>.
- [31] ruby.git - The Ruby Programming Language. <https://git.ruby-lang.org/ruby.git>.
- [32] The Python programming language. <https://github.com/python/cpython>.
- [33] The Web framework for perfectionists with deadlines | Django. <https://www.djangoproject.com/>.
- [34] Web Service Efficiency at Instagram with Python - Instagram Engineering. <https://instagram-engineering.com/web-service-efficiency-at-instagram-with-python-4976d078e366>.

- [35] What Powers Instagram: Hundreds of Instances, Dozens of Technologies. <https://instagram-engineering.com/what-powers-instagram-hundreds-of-instances-dozens-of-technologies-adf2e22da2ad>.
- [36] Adam D'Angelo. Why did Quora choose Python for its development? <https://www.quora.com/Why-did-Quora-choose-Python-for-its-development-What-technological-challenges-did-the-founders-face-before-they-decided-to-go-with-Python-rather-than-PHP>, Sep. 2014.
- [37] Alex Martelli. Heavy usage of Python at Google? <https://stackoverflow.com/questions/2560310/heavy-usage-of-python-at-google>, Apr. 2010.
- [38] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. Opus: Online patches and updates for security. In *USENIX Security Symposium*, pages 287–302, 2005.
- [39] Jeff Arnold and M Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 187–198. ACM, 2009.
- [40] Andrew Baumann, Jonathan Appavoo, Robert W Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *USENIX Annual Technical Conference*, pages 337–350, 2007.
- [41] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *USENIX Annual Technical Conference, General Track*, pages 279–291, 2005.
- [42] Eric A Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.
- [43] Derek Bruening and Saman Amarasinghe. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering . . . , 2004.
- [44] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.
- [45] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 35–44, 2006.
- [46] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. Polus: A powerful live updating system. In *29th International Conference on Software Engineering (ICSE'07)*, pages 271–281. IEEE, 2007.
- [47] Python Software Foundation. The python profilers. <https://docs.python.org/3.5/library/profile.html>, 2019.
- [48] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and automatic live update for operating systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13*, page 279–292, New York, NY, USA, 2013. Association for Computing Machinery.
- [49] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and automatic live update for operating systems. *SIGPLAN Not.*, 48(4):279–292, March 2013.
- [50] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.
- [51] Quinta group. Python at google. <https://quintagroup.com/cms/python/google>.
- [52] unicorn. Potential http response splitting vulnerability. <https://github.com/benoitc/unicorn/issues/1227>, 2016.
- [53] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Transactions on Software engineering*, 22(2):120–131, 1996.
- [54] Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for c. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12*, page 249–264, New York, NY, USA, 2012. Association for Computing Machinery.
- [55] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI '01*, page 13–23, New York, NY, USA, 2001. Association for Computing Machinery.
- [56] Gisli Hjalmtýsson and Robert Gray. Dynamic c++ classes—a lightweight mechanism to update code in a running program. In *USENIX Annual Technical Conference*, volume 98, 1998.

- [57] Information Technology Intelligence Consulting Corp. ITIC 2020 Global Server Hardware, Server OS Reliability Report. <https://www.ibm.com/downloads/cas/DV0XZV6R>, April 2020.
- [58] Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. *The art of the metaobject protocol*. MIT press, 1991.
- [59] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *European Conference on Object-Oriented Programming*, pages 327–354. Springer, 2001.
- [60] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.
- [61] Mirumee Labs. A modular, high performance, headless e-commerce storefront built with python, graphql, django, and reactjs. <https://saleor.io/>, 2020.
- [62] 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. *Acm sigplan notices*, 40(6):190–200, 2005.
- [63] Kristis Makris and Rida A Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *USENIX annual technical conference*, volume 2009. San Diego, CA, 2009.
- [64] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 327–340, 2007.
- [65] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. Disl: a domain-specific language for bytecode instrumentation. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 239–250, 2012.
- [66] Sebastien Martinez, Fabien Dagnat, and Jérémy Buisson. Pymoult : On-Line Updates for Python Programs. In *ICSEA 2015*, 2015.
- [67] James Mickens. Rivet: browser-agnostic remote debugging for web applications. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 333–345, 2012.
- [68] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for c. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '06*, page 72–83, New York, NY, USA, 2006. Association for Computing Machinery.
- [69] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for c. *ACM SIGPLAN Notices*, 41(6):72–83, 2006.
- [70] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.
- [71] Angela Nicoara, Gustavo Alonso, and Timothy Roscoe. Controlled, systematic, and efficient code replacement for running java programs. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 233–246, 2008.
- [72] NVD. Cve-2018-1000164 detail. <https://nvd.nist.gov/vuln/detail/CVE-2018-1000164>, 2018.
- [73] NVD. Cve-2018-1000164 detail. https://owasp.org/www-community/attacks/HTTP_Response_Splitting, 2018.
- [74] odoo. [payment_paypal] 500 error when process order. <https://github.com/odoo/odoo/issues/39406>, 2019.
- [75] odoo. [13.0] performance issue on validating receipts. <https://github.com/odoo/odoo/issues/46900>, 2020.
- [76] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09*, page 87–102, New York, NY, USA, 2009. Association for Computing Machinery.
- [77] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. Mvedsua: Higher availability dynamic software updates via multi-version execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 573–585. ACM, 2019.
- [78] Luís Pina, Luís Veiga, and Michael Hicks. Rubah: Dsu for java on a stock jvm. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '14*,

- page 103–119, New York, NY, USA, 2014. Association for Computing Machinery.
- [79] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: efficient dynamic weaving for java. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 100–109, 2003.
- [80] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, 2002.
- [81] Romain Komorn. Python in production engineering. <https://engineering.fb.com/production-engineering/python-in-production-engineering/>, May 2016.
- [82] Florian Rommel, Christian Dietrich, Peng Huang, Daniel Friesel, Sangeetha Abdu Jyothi, Karan Grover, Marcel Köppen, Nina Narodytska, Muthian Sivathanu, Christoph Borchert, et al. From global to local quiescence: Wait-free code patching of multi-threaded processes. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 651–666, 2020.
- [83] saleor. Category index renders extremely slow when there are many discounts. <https://github.com/mirumee/saleor/issues/1314>, 2017.
- [84] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous deployment at facebook and oanda. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 21–30. IEEE, 2016.
- [85] similartech. Market share and web usage statistics of Django. <https://www.similartech.com/technologies/django>, Jan. 2020.
- [86] Craig AN Soules, Jonathan Appavoo, Kevin Hui, Robert W Wisniewski, Dilma Da Silva, Gregory R Ganger, Orran Krieger, Michael Stumm, Marc A Auslander, Michal Ostrowski, et al. System support for online reconfiguration. In *USENIX Annual Technical Conference, General Track*, pages 141–154, 2003.
- [87] Statista Inc. Average cost per hour of enterprise server downtime worldwide. <https://www.statista.com/statistics/753938/worldwide-enterprise-server-hourly-downtime-cost/>, 2021.
- [88] Guy Steele. *Common LISP: the language*. Elsevier, 1990.
- [89] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '05*, page 183–194, New York, NY, USA, 2005. Association for Computing Machinery.
- [90] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: A vm-centric approach. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09*, page 1–12, New York, NY, USA, 2009. Association for Computing Machinery.
- [91] Suriya Subramanian, Michael Hicks, and Kathryn S McKinley. Dynamic software updates: a vm-centric approach. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2009.
- [92] Davy Suvéé, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, 2003.
- [93] Wei Tang and Min Zhang. Pyreload: Dynamic updating of python programs by reloading. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 229–238. IEEE, 2018.
- [94] thenewstack.io. Instagram Makes a Smooth Move to Python 3. <https://thenewstack.io/instagram-makes-smooth-move-python-3/>, Jun. 2017.
- [95] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: diagnosing production run failures at the user’s site. *ACM SIGOPS Operating Systems Review*, 41(6):131–144, 2007.
- [96] Wikipedia. Youtube. <https://en.wikipedia.org/wiki/YouTube>, 2020.

RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing

Mingzhe Wang
Tsinghua University

Jie Liang
Tsinghua University

Chijin Zhou
Tsinghua University

Yu Jiang
Tsinghua University

Rui Wang
Capital Normal University

Chengnian Sun
Waterloo University

Jiaguang Sun
Tsinghua University

Abstract

Coverage-guided fuzzers use program coverage measurements to explore different program paths efficiently. The coverage pipeline consists of runtime collection and post-execution processing procedures. First, the target program executes instrumentation code to collect coverage information. Then the fuzzer performs an expensive analysis on the collected data, yet most program executions lead to no increases in coverage. Inefficient implementations of these steps significantly reduce the fuzzer’s overall throughput.

In this paper, we propose RIFF, a highly efficient program coverage measurement mechanism to reduce fuzzing overhead. For the target program, RIFF moves computations originally done at runtime to instrumentation-time through static program analysis, thus reducing instrumentation code to a bare minimum. For the fuzzer, RIFF processes coverage with different levels of granularity and utilizes vector instructions to improve throughput.

We implement RIFF in state-of-the-art fuzzers such as AFL and MOpt and evaluate its performance on real-world programs in Google’s FuzzBench and fuzzer-test-suite. The results show that RIFF improves coverage measurement efficiency of fuzzers by $23\times$ and $6\times$ during runtime collection and post-execution processing, respectively. As a result, the fuzzers complete 147% more executions, and use only 6.53 hours to reach the 24-hour coverage of baseline fuzzers on average.

1 Introduction

Fuzzing is an automated testing technique that attempts to detect bugs and vulnerabilities in programs [1, 3, 9, 13, 14, 24, 27, 31, 35, 36]. Coverage-guided fuzzing improves bug-detection ability of fuzzers by leveraging program coverage measurements to guide fuzzing towards exploring new program states [4, 8, 20, 29, 40]. These fuzzers perform the following steps: ① the fuzzer selects an input from the corpus and performs mutation operations to generate new inputs; ②

the fuzzer executes the target program with mutated inputs and collects coverage statistics of these runs; ③ the fuzzer saves the input to the corpus if it can trigger bugs or find new program states. With proper coverage guidance, fuzzers can improve their efficiency by prioritizing mutation on *interesting* inputs in the corpus and discarding inputs that do not reach any new program states.

Generally speaking, the coverage pipeline of fuzzers consists of two stages: runtime coverage information collection and post-execution processing: first, the target program is instrumented with coverage collection code, which updates an array of counters to record the runtime execution trace; after the completion of an execution, the fuzzer processes the values in the array to check whether each execution reaches any new program states.

An instrumented program executes many more instructions compared to a non-instrumented binary. Since fuzzers continuously execute random inputs, a slight slow-down can significantly impact overall fuzzing performance. We analyze the source of overhead using many microarchitectural performance counters.

For the target program, fuzzers insert instrumentation code for coverage collection at each basic block. The collection code saves the current register context, loads the base address for the counter region, computes the counter index, updates the corresponding counter value, restores the context, and transfers control back to the program logic (see Figure 2). The code is executed frequently, and can contain dozens of instructions encoded in around a hundred bytes. Furthermore, modern processors use a multi-tier cache subsystem to reduce memory latency. Because the collection code updates the counter array, it adds many loads or stores to the instruction stream. These memory accesses stress the memory subsystem by competing with the program logic for instruction cache. The extra memory latency reduces the overall execution speed of programs.

For the fuzzer itself, the instructions which process coverage do not uncover new states in most cases. While new program states are extremely rare, fuzzers need to perform the

following operations: convert the raw coverage information into features, then check the database of known features, and update the database to add the newly discovered features [42]. This algorithm is implemented using memory write-, integer comparison- and conditional branching instructions. The complex nature of the code prevents the compiler from optimizing it. Consequently, the instructions emitted by the compiler cannot fully utilize instruction-level parallelism supported by the processor’s execution engine.

In this paper, we propose RIFF to reduce instruction footprints of coverage pipelines and improve fuzzing throughput. RIFF utilizes compiler analyses and leverages low-level features directly exposed by the processor’s instruction set architecture. Specifically, ① RIFF reduces the amount of instructions executed for *runtime coverage collection* in the target program. First, RIFF removes edge index computations at runtime by pre-computing the edge transfers at instrumentation-time. Next, RIFF eliminates the instructions for loading the base of the counter region by assigning the region a link-time determined static address. Thus, RIFF can use only one instruction encoded in 6 bytes per instrumentation site. ② RIFF removes unnecessary instructions when *processing coverage* in fuzzers by dividing processing granularity into three stages, where the first stage handles simple scenarios fast, while the last stage is suited for more sophisticated scenarios. For the most common case, RIFF scans the coverage region and skips zero-valued chunks using vector instructions, analyzing 16, 32, or 64 counters per iteration on modern processors.

To demonstrate the effectiveness of our approach, we implement RIFF by augmenting state-of-the-art fuzzers such as AFL [40] and MOpt [29] and evaluate its performance on real-world programs from Google’s fuzzer-test-suite [21] and FuzzBench [30]. On the coverage collection side, RIFF reduces the average runtime overhead of instrumentation from 207% to 8%. On the post-execution processing side, RIFF reduces coverage processing time from 217 seconds to 42 seconds with AVX2 instructions [10] and 31 seconds with AVX512 instructions [34]. As a result, the enhanced fuzzers can complete 147% more executions during the 24-hour experiments, covering 13.13% more paths and 5.60% more edges. Alternatively, the improved fuzzers need only 6.53 hours to reach the 24-hour coverage of baseline fuzzers.

In summary, this paper makes the following contributions:

- We observe that the collection and processing of program coverage measurements significantly affect the speed of fuzzing. We break down the cost of instrumentation and analysis code.
- We eliminate much of the runtime cost by using precomputing information statically, and we accelerate post-execution processing using vectorization.
- We adapt RIFF to popular fuzzers and achieve significant speedup on real-world programs. The coverage

analysis algorithm of our work has been integrated into production-level fuzzer AFL++ [23].

2 Background

2.1 Stages of a Coverage Pipeline

To guide fuzzing using coverage, fuzzers use a multi-stage pipeline. Figure 1 takes AFL as an example to demonstrate how fuzzers handle coverage:

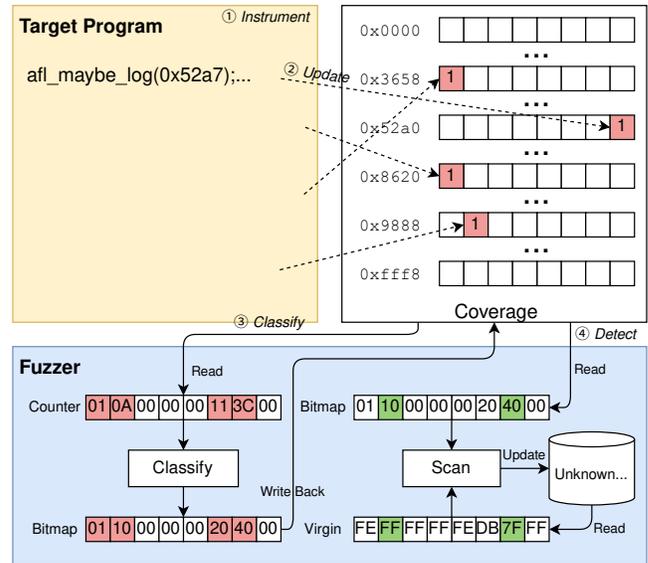


Figure 1: The coverage pipeline of the standard fuzzing tool AFL. After collecting the coverage from the target program (arrows labeled “instrument” and “update”), the fuzzer determines whether the input triggers new program behavior (“classify” and “detect”).

① *Instrument*. At compile time, `afl-clang` allocates an array of 65,536 counters to store coverage as 8-bit counters. For each basic block of the target program, `afl-as` generates a random number ID as its identifier, then inserts a call to `afl_maybe_log(ID)` at the beginning.

After instrumentation, the fuzzer generates random inputs and executes the program on each input. For each input the fuzzer detects whether the input triggers new program states by using a database, as follows:

② *Update*. At run time, `afl_maybe_log` updates the coverage counters to collect edge coverage. The logging function hashes the identifier of the previously executed and the current block, then uses the hash as an index into the counter array to increment the pointed counter by one.

③ *Classify*. After the target program completes execution, AFL reads the coverage counters to classify them into a bitmap of features. Each 8-bit counter with nonzero value is mapped to 8 possible features. The features are represented

as a bitmap, where each feature corresponds to one of the 8 bits inside the 8-bit counter. The classified result is written back to the coverage region.

④ *Detect*. With the edge transfer counts classified as a bitmap, AFL scans the database of unknown program states to detect new program behaviors: if a previously-unknown edge transfer is triggered, then the input will be labeled as “new coverage”; if a known edge transfer has different features, then it will be marked as a “new path”; otherwise, the current input is discarded. After the scan, AFL removes the newly discovered features by updating the database.

2.2 Variants of Coverage Pipeline

While the implementation varies for different fuzzers, the design mostly follows the classic coverage pipeline first introduced by AFL. Table 1 presents the instrumentation mechanism for popular fuzzers. Despite different tool chains and compiler infrastructures, all the collection methods insert code or callbacks to collect coverage. For example, although SanitizerCoverage contains a set of instrumentation options and is implemented in both Clang [7] and GCC [6], it uses callbacks and array updates to report coverage. Note that FuzzBench implements its own instrumentation for AFL [15], we only list it for completeness.

Table 1: Methods for Collecting Coverage

Method	Target	Infrastructure
afl-{clang,gcc}	Assembler	N/A
afl-clang-fast	Clang	LLVM Pass
afl-fuzzbench	Clang	SanitizerCoverage
libFuzzer	Clang	SanitizerCoverage
honggfuzz	Clang/GCC	SanitizerCoverage
Angora	Clang	LLVM Pass

Table 2 summarizes post-processing methods of coverage counters at fuzzers’ side. *honggfuzz* is a special case because it processes coverage in real-time. Other fuzzers first classify the counter array to a bitmap of features, then scan the bitmap to detect the presence of new features.

Table 2: Methods for Processing Coverage

Method	Classify	Scan
AFL	Batch	Bit twiddling
libFuzzer	Per Counter	Statistics update
honggfuzz	N/A	N/A
Angora	Distill	Queued

For example, AFL implements a two-pass design. In the first pass, it performs bitmap conversion in batch; in the second pass, it applies bit twiddling hacks for acceleration. *libFuzzer* employs a one-pass design: for each non-zero byte,

libFuzzer converts it to a feature index, then updates the local and global statistics with complex operations such as binary search. *Angora* takes the queued approach: in the first pass, it distills a small collection of counter index and feature mask out of the original array; in the second pass, it scans the collection to detect new coverage and pushes the modifications to the write-back queue; in the third pass, it locks the global database and applies the queued modifications.

3 Measuring Coverage Pipeline Overheads

To measure the overhead of the coverage pipeline, we select the classic fuzzer AFL as an example: as the forerunner of coverage-guided fuzzing, most coverage-guided fuzzers partially or completely inherit its design. As for the target program and workload, we use libxml2 from FuzzBench.

3.1 Cost of Instrumentation

To evaluate the overhead of coverage collection, we select all instrumentation methods provided by AFL, which cover all compiler infrastructures listed in Table 1. To have a fair comparison, we select afl-clang, afl-fuzzbench, and afl-clang-fast, because they have the same coverage update method and base compiler. We further decrease the optimization level of afl-fuzzbench to `-O2` to match with the other instrumentation methods.

We collect performance metrics by running the target program using perf tools. To remove the one-time cost of program startup, we do a warm-up run of the program with 1 input, then use 11 more inputs separately, then calculate the average of per-execution cost. The Intel Intrinsics Guide [12] is used as the XML input.

Table 3 lists the overhead of each collection method by normalizing each metric to the non-instrumented baseline program. Looking at the “duration” column, we can see that the instrumentation significantly slows down program execution. For example, as soon as the fastest method, afl-clang-fast, finishes executing its first input, the non-instrumented program has executed more than half of the second input.

Table 3: Overheads of Instrumented Programs

Method	Duration	Instructions	L1-I	L1-D	μ ops
afl-clang	3.50x	4.26x	102.36x	5.16x	4.72x
afl-fuzzbench	2.45x	2.83x	19.88x	2.53x	2.14x
afl-clang-fast	1.69x	1.79x	33.58x	2.88x	2.11x

The “instruction count” column explains the slowdown. Figure 2 lists the instructions of afl-clang (the slowest method), and afl-clang-fast (the fastest method). Take afl-clang for example, for each basic block, it inserts 10 instructions encoded in 56 bytes. These instructions save the context, invoke `__afl_maybe_log`, and restore the context. In

__afl_maybe_log, instructions totaling 44 bytes are executed, which update the last code location and increase the counter. They even contain a conditional jump which checks whether the coverage counters are initialized. The same problem is still applicable to the fastest method afl-clang-fast: of all the 7 instructions it executes, only one instruction is used to actually update the counter.

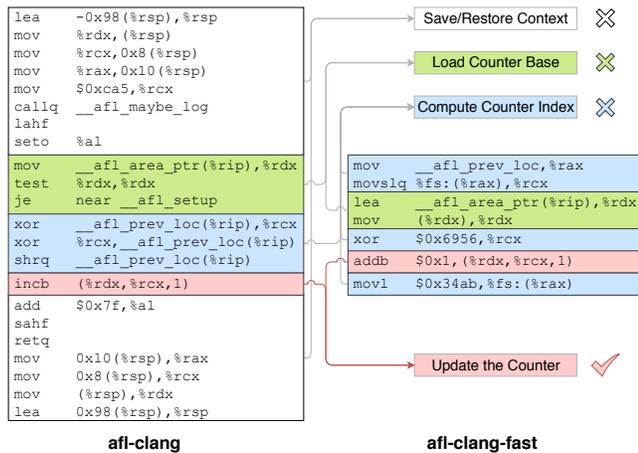


Figure 2: Instructions inserted by afl-clang (22 instructions, 100 bytes) and afl-clang-fast (7 instructions, 39 bytes). Note that only the instruction marked in red updates the counter.

The instrumentation code has a significant processor cost. First, it starves the processor’s front end which translates instructions to micro-ops. For each basic block, afl-clang requires executing extra instructions totaling 100 bytes, i.e. 1/256 of all the available L1 instruction cache. As a result, afl-clang experiences 101.36x more L1 instruction cache misses, and the CPU executes 3.72x more micro-ops for afl-clang produced programs.

3.2 Unnecessary Instructions in Fuzzer

Figure 3 presents the cost breakdown of afl-fuzz by sampling its CPU usage. To reduce noise introduced by fuzzing, we sample the user space CPU cycles for 5 seconds after afl-fuzz has discovered 2,000 paths of libxml2.

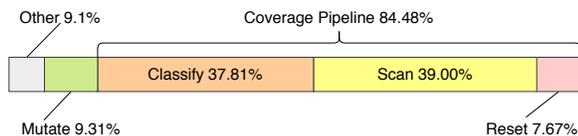


Figure 3: Breakdown of execution costs for afl-fuzz: AFL first mutates the input; after the execution completes, it classifies the coverage to bitmap, scans the bitmap for new coverage, and resets the memory for the next run.

From the figure we can see that afl-fuzz spends the majority

of its time on the coverage pipeline. To detect new program states, AFL spends 84.48% of its valuable CPU time on the coverage pipeline. The overwhelmingly high percent of CPU usage implies significant problems behind the overall system design, which inevitably leads to redundancy in executed instructions.

Table 4 shows that most executions do not improve coverage. We call a counter “useless” if its value is zero, since a zero-valued counter never maps to a feature. We call a program execution “useless” if its bitmap does not contain any new feature with respect to previous executions. After AFL terminates, we collected the coverage of the first discovered 2,000 paths, and calculated useless counters (see the first row). We also compute useless executions during a 5-second time interval (see the second row). During the period, AFL had executed 67,696 inputs, where each execution required processing 64 KiB of coverage. Although it had processed over 4,231 MiB of coverage, it only discovered 2 new paths, and none of the paths covered new counters.

Table 4: Number of Processed Counters and Executions

	Total	Useless	Proportion
Counter	65,536	64,664.37	98.67%
Execution	67,696	67,694	99.997%

As the first row in Table 4 shows, for the coverage of the first 2,000 discovered paths, 98.67% of the processed counters were zero. In other words, executing an input only covers 871.69 counters, yet the total number of counters allocated by AFL was 65,536. Angora’s instrumentation technique suffers even more, because it allocates 1MiB of memory to store coverage. The sparsity of the coverage array implies that skipping zero counters quickly during coverage analysis can be a major performance boost.

As the second row in Table 4 shows, although 99.997% of the inputs did not trigger any new program behavior, AFL still performed many computations: the first pass converted the coverage to a bitmap, and the second pass re-read it to compare with the database of unknown program states. The same applies to libFuzzer, which maintains even more statistics, including the minimum input size, the trigger frequency of each feature, the list of the rarest features, and the list of unique features covered by the current input. The analysis requires complex computations involving table lookups, linear searches, and floating-point logarithms.

The analysis logic cannot be efficiently optimized by compilers. The high-level algorithm is scattered with side effects, control-flow transfers, and data dependencies. Due to the complexity of the analysis logic, the compiler cannot perform important optimizations such as hardware-assisted loop vectorization. Only shallow optimizations, such as loop unrolling, are performed.

4 Design of RIFF

Figure 4 presents the overall design of RIFF. Similar to conventional coverage pipelines, it consists of compile-time instrumentation, runtime execution trace collection, and coverage processing.

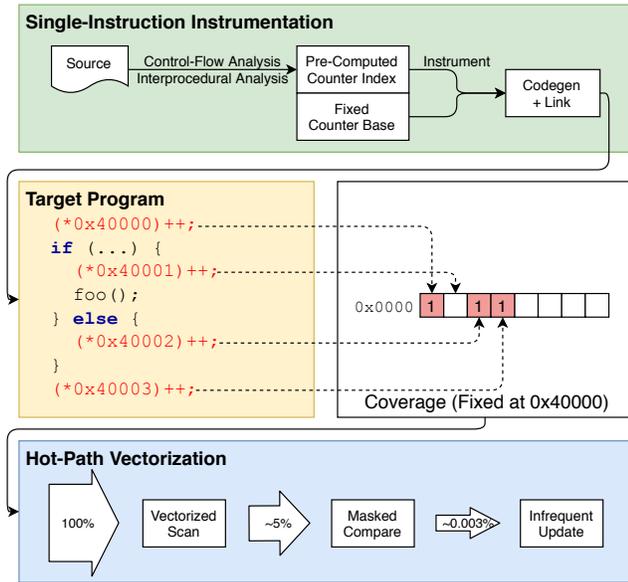


Figure 4: System overview. RIFF first instruments the program to log the execution trace at runtime. After the completion of execution, the fuzzer processes the coverage in three stages, using vectorization on the hot-path.

At compile-time, RIFF performs control-flow analysis and interprocedural analysis to pre-compute all possible control-flow edges; each edge is statically allocated a fixed counter index. The compile-time computation avoids performing the address computation at runtime. Next, RIFF inserts code to log the edge execution by incrementing the counter at the corresponding address. Finally, RIFF generates machine code with the help of the compiler’s backend, without requiring runtime context saving or restoring. When the target program starts, RIFF’s runtime maps the coverage counters at the fixed address specified by the compiler. The simplified instrumentation and aggregated coverage layout reduces the overhead of coverage collection. We describe the optimized instrumentation in detail in Section 4.1.

After the target program completes the execution of an input, the fuzzer enhanced with RIFF processes the coverage in three stages, where the first stage handles simple cases quickly, and the last stage handles infrequent complex cases. According to the simulation of collected coverage in Section 3.2, the first stage vectorized scan can eliminate 95.08% of the analysis cases, leaving only 4.92% of the processing job to the second stage, i.e. the masked comparison. The slowest stage only processes 0.003% of all cases.

4.1 Single-Instruction Instrumentation

As shown in Figure 2, the instrumentation code that collects coverage is expensive. Not only does it need to update the counter for each basic block, but the instrumentation code *saves and restores registers* around each counter update to preserve program logic. Moreover, the code *loads the counter base address* dynamically and *computes the counter index* by hashing the block index. RIFF reduces this code to a single instruction by performing much of this computation at compile time.

Pre-Compute Counter Index AFL uses hashing-based control-flow edge coverage. While edge-level coverage can distinguish between execution traces where block-based coverage cannot, maintaining the previous block’s identifier dynamically and computing hashes at runtime is expensive. Using the compiler infrastructure RIFF performs edge-coverage computation at compile-time, and falls back to runtime computation only if static information is insufficient.

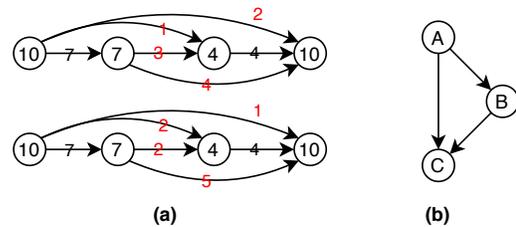


Figure 5: Problems behind raw block coverage: (a) incompleteness: multiple edge counts can map to the same block count; (b) complexity: obtaining the hit count of edge AC requires extra computation at fuzzer’s side.

Figure 5 illustrates the imprecision of block-level coverage. Figure (a) shows two control-flow graphs that have different edge counts but identical block counts. In theory, for a digraph with $|V|$ vertices, there can be $\frac{|V||V-1|}{2}$ edges. Therefore, block count alone cannot determine the exact edge counts. However, in practice, the graph is very sparse, and in some cases, the edge counts can be uniquely determined by the block counts. However, calculating edge counts requires an expensive computation to solve a system of equations. As Figure (b) shows, there are three basic blocks (A, B, and C) and three edges (AB, BC, and AC). Suppose that the instrumentation scheme collects the count for basic block A, B, and C as a , b , and c respectively. While the hit count for edge AB and BC can be directly represented as a or c , the hit count of edge AC must be computed (such as $a - b$). Solving the system of equations will significantly slow down the processing at fuzzer’s side.

RIFF leverages static analysis to allocate one counter for each edge. It does this by creating additional empty basic blocks when needed. As Algorithm 1 shows, if the hit count of an edge can be uniquely determined by its source or sink

Algorithm 1: Control-Flow Edge Instrumentation

Data: A control-flow graph $G = (V, E)$
Result: A new control-flow graph $G' = (V', E')$ and a set of target blocks to instrument $T \subseteq 2^{V'}$

```
 $T \leftarrow \emptyset, V' \leftarrow V, E' \leftarrow E;$   
for  $(x, y) \in E$  do  
  if  $\delta^+(x) = 1$  then  
    The source vertex has only one outgoing edge  $(x, y)$ ,  
    thus the hit count of  $(x, y)$  equals to  $x$ ;  
     $T \leftarrow T \cup x;$   
  else if  $\delta^-(y) = 1$  then  
    The sink vertex has only one incoming edge  $(x, y)$ ,  
    thus the hit count of  $(x, y)$  equals to  $y$ ;  
     $T \leftarrow T \cup y;$   
  else  
    No direct representation is available ;  
    Introduce a temporary vertex  $t_{(x,y)}$  to represent the  
    hit count of  $(x, y)$  ;  
     $V' \leftarrow V' \cup t_{(x,y)};$   
     $E' \leftarrow E' / (x, y) \cup (x, t_{(x,y)}) \cup (t_{(x,y)}, y);$   
     $T \leftarrow T \cup t_{(x,y)};$   
  end  
end
```

vertex, then the block count is used for the edge. Otherwise, an empty block is allocated to represent the edge. For function calls, RIFF uses the hit count of the caller block to represent the hit count for the edge between the caller block and the callee’s entry block because the counts are equal. After collecting the blocks to instrument, RIFF assigns identifiers sequentially for each block and removes instrumentation sites whose hit counts can be represented by other counters. These identifiers are used as runtime indexes for the counters in the coverage array.

Fix Counter Base AFL uses a block of shared memory for the counters. When the target program starts, the runtime library maps the shared memory into its address space and stores the base address as a global variable. While indirect addressing is flexible, computing the counter address dynamically for every basic block is inefficient. To remove extra accesses to the counter base, the address must be compile-time constants for each instrumentation site. Counter allocation with fixed addresses is done in two steps.

At the beginning of each basic block, the instrumentation code should increment its associated counter. If the base address is fixed, and the index of the array is already allocated at compile time (see Section 4.1), the address of the counter can be also computed at compile time. We can then directly increment the counter pointed by the address, using e.g., `incb $ADDR`. However, as Table 5 shows, directly encoding the target address inside the instruction requires a 7-byte instruction (scale-index-base). RIFF uses a RIP-based addressing mode

¹, requiring a 6-byte instruction. Moreover, the expensive register save/restore code is no longer needed.

Table 5: Instruction Encoding of Addressing Modes

Assembly	Length	Opcode	ModRm	SIB	Disp
<code>incb \$ADDR</code>	7	0xfe	0x04	0x25	(4 bytes)
<code>incb \$OFFSET(\$rip)</code>	6	0xfe	0x05		(4 bytes)

Before the target program runs, the memory shared by the fuzzer should be correctly mapped to the address space of the target program. To prevent the static and dynamic linker from reusing the address for other symbols, RIFF fixes the binary’s image base to the address 0x80000 (8 MiB), and reserves the address range of 0x400000 to 0x80000 for the coverage.

Indirect Control Transfers While single-instruction instrumentation is efficient, this solution cannot be used for indirect control transfers. These occur in the following instances: GNU C extensions that allow taking the address of switch labels [18], `set jmp` and `long jmp` [26], function pointers, and unwinds on C++ exceptions.

RIFF uses interprocedural control-flow analysis to discover such cases and falls back to dynamic computation. If the start of an edge representing indirect control transfer is found (e.g. `set jmp`), RIFF stores the source block ID in thread-local storage before performing the transfer. At the target of an indirect control transfer (e.g. `long jmp`), RIFF loads the source block ID and computes the counter index by hashing.

4.2 Hot-Path Vectorized Analysis

As Table 4 shows, among all coverage counters, only a small number of counters are updated by the target program; among all executions, inputs which demonstrate new program behavior are extremely rare. This observation implies that many computations performed by the fuzzer do not produce useful results. If the redundant computation is removed, the simplified logic can be accelerated using SIMD instructions (Advanced Vector Extensions on x86-64 and NEON on ARMv8).

Figure 6 demonstrates how this multi-stage processing design simplifies the logic. Stage 0 is the simplest one, which just fetches 64 bytes chunks and discards all-zero chunks. Stage 1 is invoked with the nonzero positions encoded as a mask. In Stage 1 the counters are classified as a bitmap in registers then directly compared with the database for unknown program states. The counters are discarded if no new features are discovered. Only when it is determined that the current input triggers new program behaviors is the original analysis performed by AFL invoked, in Stage 2. While this stage requires complex computations, it is rarely invoked.

¹RIP is the instruction pointer.

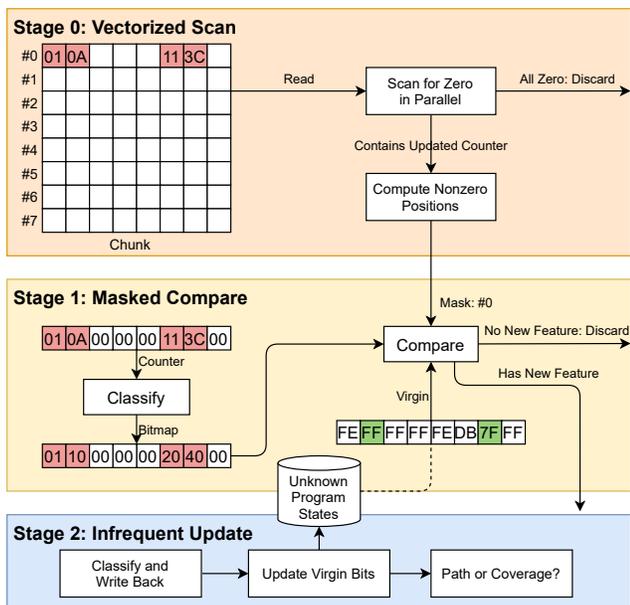


Figure 6: Processing coverage in three stages. Stage 0 filters out large chunks of zero bytes; Stage 1 checks for new coverage using masked comparisons; Stage 2 is invoked only for inputs that trigger new behaviors.

Vectorized Scan Although coverage-guided fuzzing can discover lots of code during the whole fuzz session, the covered code for a single input is lower. Because most counters are not accessed by the target program, their values stay zero after execution. Filtering out these zero counters can remove further processing stages, but the filtering operation itself requires extra computation.

To filter the zero counters efficiently, we use instructions that scan counters in parallel. Modern processors have vector processing abilities. AVX512 is a typical single-instruction-multiple-data design proposed by Intel in 2013, and it is widely supported in modern server processors. Operating on 512-bit vectors, it can compare 64 lanes of 8-bit integers in parallel (`vptestnmb`). For example, on Skylake-X based processors, it completes such a scan in 4 clock cycles. By comparison, the scalar-based processing requires 64 `testb` instructions with a latency of 1 cycle each.

The vectorized comparison encodes the comparison results inside a mask register. Each bit inside the mask register represents whether a lane inside the vector is zero. For example, if we treat 64 bytes of data chunk as 8 lanes of u64, then the result mask register contains 8 bits. If the least significant bit (0x1) is set in the mask, then the first (#0) lane is zero. Similarly, if the most significant bit (0x80) is set in the mask, then the last (#7) lane is zero. Consequently, we can skip the following tiers if all the 8 bits are set (0xff), indicating that all the lanes are zero.

Masked Compare If a chunk contains non-zero bytes, it may represent a new program behavior. Therefore, vectorized scan cannot discard the chunk and should delegate the computation to the next stage, masked compare. In this stage, the coverage is classified then compared with the database to detect new program behavior.

However, even for a non-zero chunk, it is very likely that most of the lanes are zero because of the sparsity of coverage. To remove unnecessary computation, the mask obtained from vectorized scan is used to sift the nonzero lanes: only when the mask indicates that a lane is non-zero, then the following classification is used. Otherwise, the zero lanes are discarded immediately.

For each nonzero lane, the corresponding counters are read into a register. After classifying the raw counters into bitmap using table lookups and bitwise operations, they are directly compared with the database. In most cases, the comparison will not find a difference and the bitmap is discarded. We optimize for the scenario where the bitmap is discarded to avoid updates to both the bitmap and the database.

Infrequent Update For inputs triggering interesting behavior, the processing of its coverage will reach stage 2. This stage is seldom invoked.

This stage performs the original analysis performed by standard fuzzers: first, it classifies the original counters and writes the bitmap back to memory; second, it reads the bitmap, compares it with the database, and updates the database if needed. While scanning the bitmap, it checks for changed counters and declares the run to be a “new coverage” if any are found. Otherwise, the run has discovered a “new path”.

5 Implementation

Because single-instruction instrumentation requires the precise counter address for each instrumentation point, instrumentation must be performed on the whole program, at link-time. The compiler part of RIFF is implemented on LLVM. Specifically, when compiling source files, RIFF instructs the compiler to produce LLVM bitcode instead of object files. These bitcode files are linked to the whole-program bitcode for analysis use. Next, RIFF performs instrumentation on the whole program leveraging the DominatorTreeAnalysis and BasicBlockUtils analyses, then generates machine code as a single object file. During code generation, LLVM prefers to generate 7-byte `addb` instructions over the 6-byte `incb` instructions, because the default configuration of LLVM is optimized for old or mobile-first processors, where `incb` is slower than `addb`. To force instruction selection to generate `incb` instructions, we fine-tune the LLVM target attribute by disabling the `slow-incdec` target feature.

As in conventional linking, the single object file is linked with system libraries. After this step the compiler maps the

symbol denoting the start of coverage counters at a fixed address (SHN_ABS in `st_shndx`). As Listing 1 shows, the generated machine code only requires 6 bytes for most cases. Only on indirect transfers does RIFF fall back to the runtime hashing.

```
# Single-instruction instrumentation
incb $INDEX(%rip) # fe 05 ?? ?? ?? ??

# Rare case: indirect transfer (source)
mov $PREV(%rip),%rcx # 48 8b 0d ?? ?? ?? ??
movl $BBID,%fs:(%rcx) # 64 c7 01 ?? ?? ?? ??

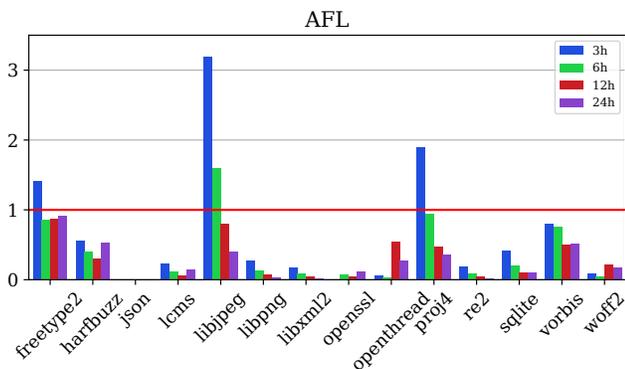
# Rare case: indirect transfer (destination)
mov $PREV(%rip),%rcx # 48 8b 0d ?? ?? ?? ??
movslq %fs:(%rcx),%rax # 64 48 63 01
xor $BBID,%rax # 48 35 ?? ?? ?? ??
incb $BASE(%rax) # fe 80 ?? ?? ?? ??
```

Listing 1: Assembly and machine code generated by RIFF.

Because vectorized coverage processing relies on the hardware support of SIMD instructions, currently we implement two variants on x86-64. If AVX512 Doubleword and Quadword Instructions (AVX512DQ) are supported, then 8 lanes of 64-bit integers are processed as a chunk. If AVX2 is supported, then the 4 lanes of 64-bit integers are processed as a chunk. Otherwise, Stage 0 is skipped entirely, and Stage 1 is executed. We implement the algorithms via intrinsic functions to take advantage of the compiler-based register allocation optimization.

6 Evaluation

To demonstrate how the reduced instruction footprint accelerates fuzzing, we evaluate the performance of RIFF on real-world settings.



For target programs, we select every program included in both Google fuzzer-test-suite and FuzzBench. Carefully picked by Google, they encompass a comprehensive set of widely-used real-world programs. For fuzzers, we select the classic industrial fuzzer AFL and the recently published MOpt.

We compile the programs with `afl-clang` using the default settings and compile RIFF’s version with our instrumentation pipeline. For both cases we use Clang 11.0 with the same configuration (e.g., optimization level). As for the fuzzers, the baseline versions are built from the git repositories without modification. We further apply RIFF’s hot-path acceleration patch to the baseline fuzzers. Note that RIFF and AFL use different instrumentation, we calibrate the raw metrics with fuzzer-test-suite’s coverage binary for fairness. All the coverage used in the following analysis is based on the calibrated data.

We perform the experiments on Linux 5.8.10 with 128 GiB of RAM. The processor used is Intel Xeon Gold 6148. Its Skylake-Server microarchitecture allows acceleration with AVX2 and AVX512.

6.1 Overall Results

Figure 7 compares the time required by RIFF to reach the same coverage as AFL and MOpt respectively running for 3h, 6h, 12h, and 24h. A bar below the red line indicates a speed-up for RIFF.

The purple bars show the speedup of the long experiments run for 24 hours, where fuzzing tends to saturate (discovering few new paths). On average, to reach the final coverage of AFL and MOpt running for 24 hours, RIFF’s improved versions only require 6.23 and 6.82 hours respectively. For individual programs, the improvements are consistent: even for the worst programs (freetype2 for AFL and libjpeg for MOpt), RIFF still reached the final coverage 2.1 and 0.8 hours before the baseline versions. On average, RIFF accelerates

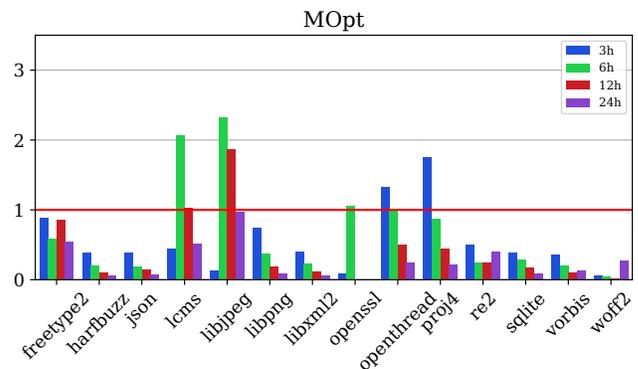


Figure 7: Normalized execution time required by RIFF to reach the same coverage as AFL and MOpt. The X axis is programs, the Y axis is the ratio between the execution times required for reaching the same coverage. A bar below the red line indicates a speed-up.

the 24-hour fuzzing by 268.74%.

The bars of 3h, 6h, and 12h show the speedup for shorter experiments. In such scenarios, saturation is less likely, and the randomness can lead to slowdowns (causing a different set of inputs to be explored). Here we can see that RIFF is still frequently performing best. For example, when fuzzing freetype2 with AFL, RIFF-based version requires 1.22 hours more to catch up with the baseline version, but its performance gradually improves as we extend the experiment, and it leads by 0.85, 1.56, and 2.10 hours at 6, 12, and 24h respectively.

Figure 8 presents the overall results after 24-hour experiments. Inside the figure, the baseline metrics from AFL and MOpt are normalized to the red horizontal line 1.0, while the corresponding metrics from RIFF’s optimizations are drawn as bars. Higher bars indicate better performance.

The “covered edges” graph from Figure 8 demonstrates the overall improvement brought by RIFF. On average, RIFF improves the coverage of AFL and MOpt by 4.96% and 6.25% respectively. The improvement is consistent for individual programs: among all the 28 experiments, RIFF is best for 27. Because RIFF accelerates both the fuzzer and the target program, more executions can be completed in less time. Despite the trend of saturation for the long 24-hour trials, RIFF still managed to cover rare edges requiring a large number of executions.

The “total paths” graph from Figure 8 demonstrates that RIFF has comparably good feedback signal as the baseline versions. For most programs, RIFF improves the total number of discovered paths since it performs more execution: on average, RIFF improves the number of discovered paths by 10.79% and 15.48% over AFL and MOpt respectively. Although RIFF simplifies the computation of edge coverage, its ability in providing fuzzing signal is not reduced because of the compile-time analysis. Take re2 for example, both the baseline versions seem to discover more paths; however, paths only provide fuzz signals, thus more paths do not necessarily lead to more coverage. When the fuzz-oriented coverage

is calibrated to fuzzer-test-suite’s canonical coverage, RIFF-based fuzzers discover more edges.

The advantage of RIFF can be seen in the “total executions” graph. RIFF increases the number of fuzzing executions in the same amount of time to values ranging between 1.03% to 541.38%. While the randomness introduced by fuzzing algorithms can cause diminished coverage, the overall result confirms that RIFF improves the execution in general. The vastly increased number of executions can be attributed to the reduced overhead, in both the target program and the fuzzer’s side.

6.2 Simplified Coverage Collection

Single-instruction instrumentation reduces the overhead of the instrumentation. To evaluate it fairly, we first fix a set of inputs, and we reuse the same inputs for all measurements for all fuzzers. For each program, we mix 1000 inputs discovered by all fuzzers; while executing the programs, we measure the time and normalize it against the non-instrumented version.

Figure 10 shows the instrumentation overhead for both afl-clang and RIFF. The figure demonstrates that the widely used instrumentation scheme afl-clang imposes heavy overhead on all the programs. Compared to the non-instrumented programs, programs instrumented by afl-clang the average execution time increases by 206.83%. The reasons can be explained by Figure 9: it executes 340.63% more instructions, which translate to 338.47% more uops and require 242.97% more L1 instruction cache refills.

RIFF reduces the footprint of instructions down to one instruction per site. On average, the coverage collection of RIFF only requires 8.40% more time to execute, while afl-clang requires 206.83% more time. In other words, RIFF reduces the overheads by 23 times. The improvement can be explained by the reduced instruction footprint: RIFF eliminates loads to counter base, shifts computation of counter index to compile-time, and removes the context saving or restoring code.

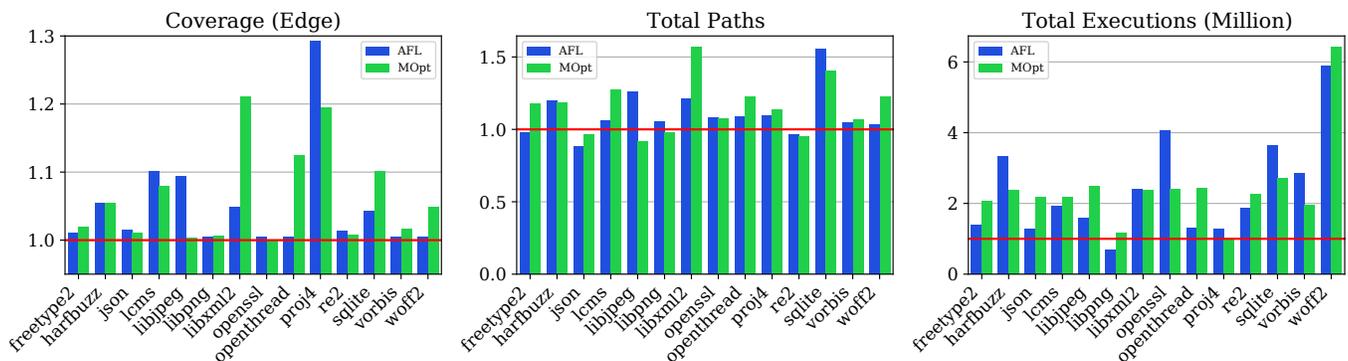


Figure 8: Normalized performance metrics for RIFF-based fuzzers after 24 hours of fuzzing. X axis is programs, Y axis is the normalized performance metric (ratio between RIFF and standard fuzzer). Bars higher than 1 (red line) indicate better performance.

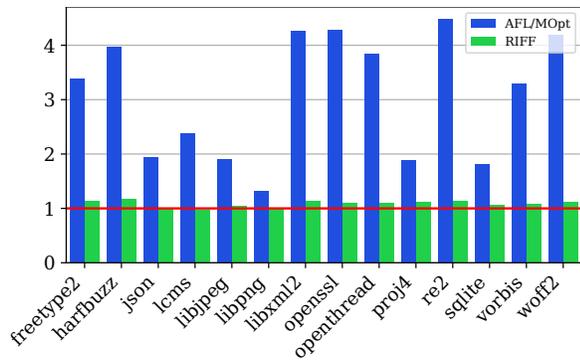


Figure 10: Normalized execution duration of fuzzed programs: time to execute 1000 on fixed inputs normalized to the time of uninstrumented programs. Lower bars indicate better performance.

6.3 Accelerated Coverage Processing

Hot-path vectorization accelerates the coverage processing at fuzzer’s side. To cancel randomness from fuzzing loops and irrelevant speedups from the target programs, we extract the coverage processing routine as a library and evaluate it in isolation.

As in Section 6.2, we fix a set of inputs, then run experiments with these inputs to collect the raw coverage counter arrays. However, because all the saved inputs are rare cases which lead to new coverage, just running coverage processing routine on the saved inputs one by one exaggerates the rate of discovery. Instead, we calculate the average number of executions to discover a new input during the whole fuzz session, and run coverage processing routine on the raw coverage repeatedly this many times on the first 50 inputs. We further calculate the total processing time required to discover the first 50 inputs; we present the normalized values in Figure 11.

Figure 11 shows the benefits of hot-path vectorization. The processing time of AFL and MOpt is normalized to 1.0, shown as the red horizontal line. The bars show the processing time of RIFF.

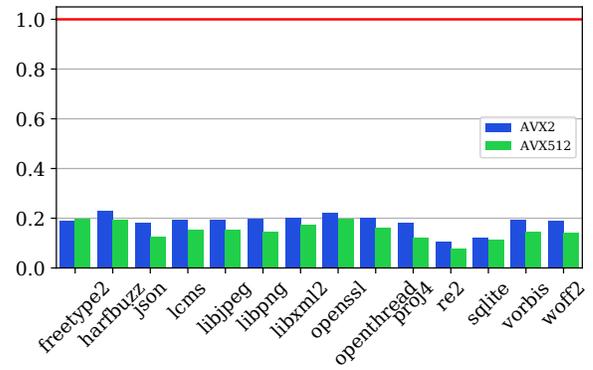


Figure 11: Coverage processing time (normalized against the baseline algorithm). Lower bars indicate better performance.

The bars for AVX2 and AVX512 of Figure 11 demonstrate RIFF’s improved efficiency in coverage processing. Leveraging AVX2, RIFF uses one instruction to compare 32 coverage counters in parallel; AVX512 further extends the parallelism to 64 counters per comparison. With hardware-assisted processing, the vectorized versions improve the efficiency of the original scalar-based pipeline by 4.64x and 6.01x respectively.

7 Discussion

Currently, we only evaluated our work on x86-64 due to insufficient fuzzer support on other platforms. For example, AFL only provides experimental ARM support via QEMU. While the implementation is target-dependent, the general idea applies to all platforms: the minimal instrumentation logic can be implemented with just 4 instructions on ARMv8 or RISC-V systems; the vectorized coverage processing can use ARMv8 NEON ISA instead of AVX2 or AVX512.

As for the applicability of our improvement, we only applied our work to the industrial fuzzer AFL and the academic work MOpt due to limited resources. While they use different fuzzing algorithms, the improvements brought by RIFF are similar (see Figure 7 and 8). Our work can be easily adapted

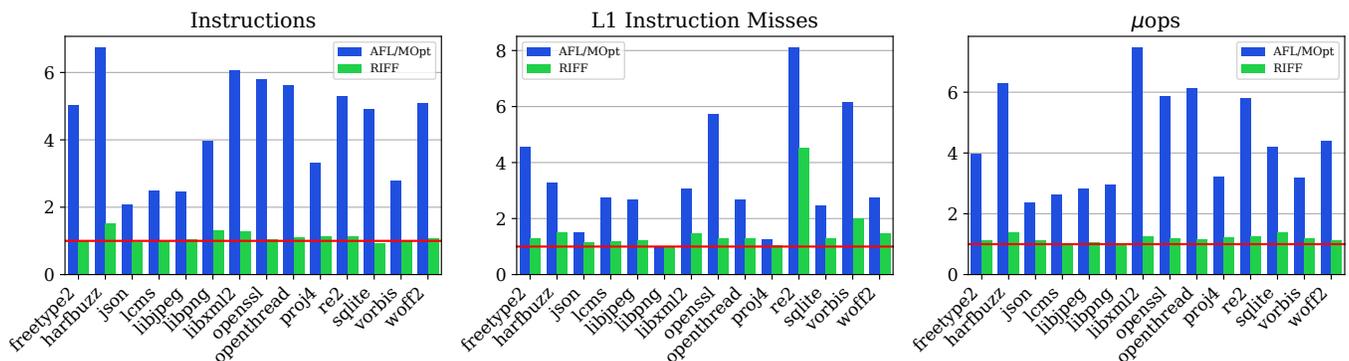


Figure 9: Metrics for RIFF-Based Programs

to more fuzzers. For example, developers of AFL++ [17] have adapted our work to their code base and conducted independent third-party evaluations with Google FuzzBench [30]. According to the result [5], our modification (labeled as “skim”) was the best-performing one among all the 10 variants.

8 Related Work

8.1 Vectorized Emulation

Snapshot fuzzing [2, 16] tests the target program from partially executed system states. The program is broken into small pieces of code, and the execution of the code is emulated by the hypervisor. Because the emulation simplifies the logic to execute, multiple system states can be emulated simultaneously with vectorization.

Rather than accelerating the emulation, RIFF is focused on coverage pipeline: first, RIFF’s single-instruction instrumentation combined with vectorization-based emulation and checkpointing accelerates the execution of target programs; RIFF’s hot-path optimization also accelerates fuzzer’s coverage analysis.

8.2 Enriching Semantics of Coverage

Since the coverage quality is crucial for input prioritization, numerous approaches have been proposed by academia which bring more semantics to coverage. For example, VUzzer [33] stores call stack information to coverage, and Angora enhances coverage with search targets [11]. Sometimes, researchers introduce data-flow features to conventional control-flow-oriented coverage. For example, Steelix [28] stores branch comparison operators, Dowser [22] records branch constraints, GreyOne [19] imports constraint conformance to tune the evolution direction of fuzzing, and [37] traces memory accesses. While these techniques can help a fuzzer to choose better inputs, the complexity introduces heavy overhead and severely limits the execution speed.

8.3 Reducing Overhead of Coverage

Not instrumenting the program eliminates overhead altogether. Researchers utilize debugger breakpoints to detect the first time a block has been covered with hardware support [32, 43]; in this scheme, only the first occurrence of a block has extra cost. However, the information of the number of times that a block has been covered is lost without any instrumentation; on the contrary, RIFF does not reduce the quality of feedback.

Another idea is to reduce the number of instrumentation points [25]. However, the cost of each instrumentation point is still high because it still needs to maintain the edge information by hashing. RIFF simplifies instrumentation points to single instructions; it is not focused on reducing the amount of instrumentation points.

8.4 Reducing Overhead of Operating System

Traditionally, fuzzing is targeted at utility programs where each execution requires `fork` a new process and then `execve` to the new binary. To remove the costly `execve`, AFL implements `fork server mode` [39]. To reduce the cost of `fork`, Xu et al. [38] designs a new system call `snapshot` to restore the execution state in-place. To further reduce the number of invocations of `fork`, AFL implements `persistent mode` [41], where a program runs continuously without restart. `libFuzzer` further eliminates other expensive system calls with in-process fuzzing: if the fuzz target is library, then the fuzzing is performed in-memory.

With these operating system works, the major overhead introduced by context switches of system calls has been greatly reduced. Consequently, the cost of execution has become another prominent problem. RIFF reduces the cost by reducing the instruction footprint of the coverage pipeline.

9 Conclusion

In this paper, we present RIFF to reduce the instruction footprint for fuzzing. We first observe that the coverage pipeline in fuzzing slows down the overall execution speed. We find that the heavy instruction footprint is the root cause: for target programs, the expensive instructions collect coverage inefficiently; for fuzzers, the unnecessary instructions cannot fully exploit the processor’s ability. We implement RIFF to reduce the instruction footprint and achieve a 268.74% speedup for the 24-hour experiments. RIFF is being integrated by popular fuzzers such as AFL and AFL++ for use in industry and has shown significant improvements over the state of the art.

Acknowledgments

We sincerely appreciate the shepherding from Mihai Budiu and Eric Schkufza. We would also like to thank the anonymous reviewers for their valuable comments and input to improve our paper. This research is sponsored in part by the NSFC Program (No. 62022046, U1911401, 61802223), National Key Research and Development Project (Grant No. 2019YFB1706200) and Ali-Tsinghua Database Testing Research Project (NO. 20212000070).

References

- [1] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. Continuous fuzzing for open source software. <https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html>, 2016. [Online; accessed 15-May-2021].

- [2] Ron Aquino. Mitigating vulnerabilities in endpoint network stacks - Microsoft Security. <https://www.microsoft.com/security/blog/2020/05/04/mitigating-vulnerabilities-endpoint-network-stacks/>, 2020. [Online; accessed 29-April-2021].
- [3] Abhishek Arya, Oliver Chang, Max Moroz, Martin Barbella, and Jonathan Metzman. Open sourcing ClusterFuzz. <https://opensource.googleblog.com/2019/02/open-sourcing-clusterfuzz.html>, 2019. [Online; accessed 15-May-2021].
- [4] Chromium Authors. libFuzzer in Chrome. <https://chromium.googlesource.com/chromium/src/+refs/heads/main/testing/libfuzzer/README.md>, 2017. [Online; accessed 15-May-2021].
- [5] FuzzBench Authors. FuzzBench: 2020-12-18 report. <https://www.fuzzbench.com/reports/experimental/2020-12-18/index.html>, 2020.
- [6] GCC Authors. Instrumentation options (using the GNU compiler collection (GCC)). <https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html>, 2021. [Online; accessed 29-April-2021].
- [7] LLVM Authors. SanitizerCoverage — Clang 12 documentation. <https://clang.llvm.org/docs/SanitizerCoverage.html>, 2021. [Online; accessed 29-April-2021].
- [8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1032–1043, 2016.
- [9] Foster Brereton. Binspector: Evolving a security tool. <http://web.archive.org/web/20181020154300/https://blogs.adobe.com/security/2015/05/binspector-evolving-a-security-tool.html>, 2015. [Online; accessed 15-May-2021].
- [10] Mark J Buxton. Haswell new instruction descriptions now available! <https://software.intel.com/content/www/us/en/develop/blogs/haswell-new-instruction-descriptions-now-available.html>, 2020. [Online; accessed 15-May-2021].
- [11] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.
- [12] Intel Corporation. Intel® Intrinsic guide. <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>, XML file available at <https://software.intel.com/sites/landingpage/IntrinsicsGuide/files/data-3.5.4.xml> in 6,096,526 bytes, 2020. [Online; accessed 15-May-2021].
- [13] Joe W. Duran and Simeon C. Ntafos. A report on random testing. In Seymour Jeffrey and Leon G. Stucki, editors, *5th International Conference on Software Engineering (ICSE)*, pages 179–183, 1981.
- [14] Michael Eddington. Peach Fuzzer: Discover unknown vulnerabilities. <http://web.archive.org/web/20210121202148/https://www.peach.tech/>, 2015. [Online; accessed 15-May-2021].
- [15] Jonathan Metzman et al. fuzzbench/fuzzer.py at master · google/fuzzbench. <https://github.com/google/fuzzbench/blob/master/fuzzers/afl/fuzzer.py>, 2020.
- [16] Brandon Falk. Vectorized emulation: Hardware accelerated taint tracking at 2 trillion instructions per second | Gamozo Labs Blog. https://gamozolabs.github.io/fuzzing/2018/10/14/vectorized_emulation.html, 2018. [Online; accessed 29-April-2021].
- [17] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In Yuval Yarom and Sarah Zennou, editors, *14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.
- [18] Inc. Free Software Foundation. Labels as values (using the GNU compiler collection (GCC)). <https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html>, 2020. [Online; accessed 15-May-2021].
- [19] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREY-ONE: data flow sensitive fuzzing. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium*, pages 2577–2594, 2020.
- [20] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy (SP)*, pages 679–696, 2018.
- [21] Google. google/fuzzer-test-suite: Set of tests for fuzzing engines. <https://github.com/google/fuzzer-test-suite>, 2020. [Online; accessed 15-May-2021].
- [22] István Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowser: A guided fuzzer for finding buffer overflow vulnerabilities. *login Usenix Mag.*, 38(6), 2013.

- [23] Marc Heuse. AFLplusplus/coverage-64.h at stable · AFLplusplus/AFLplusplus. <https://github.com/AFLplusplus/AFLplusplus/blob/stable/include/coverage-64.h>, 2021. [Online; accessed 13-May-2021].
- [24] Sam Hocevar. zzuf – Caca Labs. <http://caca.zoy.org/wiki/zzuf>, 2007. [Online; accessed 15-May-2021].
- [25] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. INSTRIM: Lightweight instrumentation for coverage-guided fuzzing. In *25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.
- [26] IEEE and The Open Group. setjmp.h - stack environment declarations. <https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/setjmp.h.html>, 2018. [Online; accessed 15-May-2021].
- [27] Cheick Keita, Marina Polishchuk, Patrice Godefroid, William Blum, Stas Tishkin, Dave Tamasi, and Marc Greisen. Microsoft security risk detection ("Project Springfield"). <https://www.microsoft.com/en-us/research/project/project-springfield/>, 2015. [Online; accessed 26-January-2018].
- [28] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *11th Joint Meeting on Foundations of Software Engineering*, pages 627–637, 2017.
- [29] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium*, pages 1949–1966, 2019.
- [30] Jonathan Metzman, Abhishek Arya, and Laszlo Szekeres. FuzzBench: Fuzzer benchmarking as a service. <https://security.googleblog.com/2020/03/fuzzbench-fuzzer-benchmarking-as-service.html>, 2020. [Online; accessed 13-May-2021].
- [31] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.
- [32] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *IEEE Symposium on Security and Privacy (SP)*, pages 787–802, 2019.
- [33] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *24th Annual Network and Distributed System Security Symposium (NDSS)*, 2017.
- [34] James Reinders. Intel® AVX-512 instructions. <https://software.intel.com/content/www/us/en/develop/articles/intel-avx-512-instructions.html>, 2017. [Online; accessed 15-May-2021].
- [35] Ari Takanen, Jared DeMott, and Charlie Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., USA, 1st edition, 2008.
- [36] Dmitry Vyukov. google/syzkaller: syzkaller is an unsupervised coverage-guided kernel fuzzer. <https://github.com/google/syzkaller>, 2015. [Online; accessed 15-May-2021].
- [37] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *27th Annual Network and Distributed System Security Symposium (NDSS)*, 2020.
- [38] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2313–2328, 2017.
- [39] Michal Zalewski. Fuzzing random programs without execve(). <http://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html>, 2014. [Online; accessed 15-May-2021].
- [40] Michal Zalewski. american fuzzy lop. <https://lcamtuf.coredump.cx/afl>, 2015. [Online; accessed 15-May-2021].
- [41] Michal Zalewski. New in AFL: persistent mode. <http://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html>, 2015. [Online; accessed 15-May-2021].
- [42] Michal Zalewski. Technical "whitepaper" for afl-fuzz. https://lcamtuf.coredump.cx/afl/technical_details.txt, 2015. [Online; accessed 15-May-2021].
- [43] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 858–870, 2020.

TCP-Fuzz: Detecting Memory and Semantic Bugs in TCP Stacks with Fuzzing

Yong-Hao Zou, Jia-Ju Bai
Tsinghua University

Jielong Zhou, Jiangfeng Tan, Chenggang Qin
Ant Group

Shi-Min Hu
Tsinghua University

Abstract

TCP stacks provide reliable data transmission in network, and thus they should be correctly implemented and well tested to ensure reliability and security. However, testing TCP stacks is difficult. First, a TCP stack accepts packets and system calls that have dependencies between each other, and thus generating effective test cases is challenging. Second, a TCP stack has various complex state transitions, but existing testing approaches target covering states instead of covering state transitions, and thus their testing coverage is limited. Finally, our study of TCP stack commits shows that 87% of bug-fixing commits are related to semantic bugs (such as RFC violations), but existing bug sanitizers can detect only memory bugs not semantic bugs.

In this paper, we design a novel fuzzing framework named TCP-Fuzz, to effectively test TCP stacks and detect bugs. TCP-Fuzz consists of three key techniques: (1) a *dependency-based strategy* that considers dependencies between packets and system calls, to generate effective test cases; (2) a *transition-guided fuzzing approach* that uses a new coverage metric named *branch transition* as program feedback, to improve the coverage of state transitions; (3) a *differential checker* that compares the outputs of multiple TCP stacks for the same inputs, to detect semantic bugs. We have evaluated TCP-Fuzz on five widely-used TCP stacks (TLDK, F-Stack, mTCP, FreeBSD TCP and Linux TCP), and find 56 real bugs (including 8 memory bugs and 48 semantic bugs). 40 of these bugs have been confirmed by related developers.

1 Introduction

The TCP protocol is a transport-layer network protocol that receives system calls and packets to provide reliable data transmission. It carries over 85% of network traffic nowadays [44, 63]. In practice, the TCP protocol has different implementations, forming different TCP stacks. Each modern operating system (such as Linux and FreeBSD) has its own kernel-level TCP stack to provide fundamental network sup-

port for user-level applications. Besides, to achieve better performance and reduce impact on OS kernels, many user-level TCP stacks (such as mTCP [26], TLDK [59] and F-Stack [19]) have been developed and widely-used in telecom systems and network nodes, to transfer data without OS involvement.

Though TCP stacks are critical, correctly implementing them is difficult [4, 16], as a TCP stack has rich functionalities (such as reliable transmission and congestion control), complex state model and various kinds of possible exceptions to handle. Thus, developers may unintentionally make mistakes when implementing TCP stacks, introducing bugs that can cause serious problems. Memory bugs (such as null-pointer dereferences and use-after-free issues) are common in TCP stacks, and they can cause crashes, data corruption and so on. Moreover, according to our study of TCP stack commits, 87% of bug-fixing commits are related to *semantic bugs* (such as RFC violations), which are related to code logics and RFC documents, instead of memory accesses. For example, CVE-2019-11478 [15] reports that the TCP retransmission queue in the Linux TCP stack can be fragmented when handling certain TCP Selective Acknowledgment (SACK) sequences, and attackers can exploit this bug to cause a denial of service. Thus, it is important to test TCP stacks to detect bugs.

To detect bugs in TCP stacks, some approaches [34, 40, 41, 53] use model checking or formal verification to check the correctness of TCP implementation. But they require much manual effort and TCP-specific knowledge to provide a complete and correct TCP state model, and they are often time-consuming due to high complexity of TCP state transitions. To reduce manual effort and time usage, some approaches [9, 30] perform static analysis of TCP stack source code. But they often introduce false positives, due to lacking exact runtime information. To reduce false positives, some approaches [3–5, 66] analyze the runtime traces of TCP stacks to infer RFC violations. However, they require substantial and effective test cases to achieve high testing coverage.

To generate effective test cases, many recent approaches perform fuzz testing for the implementations of application-layer network protocols, such as DTLS/TLS [17, 52, 54, 60],

FTP [6, 21, 43] and Modbus [37, 38]. But these approaches are limited in testing TCP stacks for three critical reasons: (1) These approaches generate just packets as input test cases, without considering dependencies between inputs; but TCP stacks receive both system calls (syscalls) and packets, which have dependencies between each other. Thus, these approaches are limited in generating effective test cases for TCP stacks. (2) These approaches use code coverage as program feedback to cover different protocol states; but besides states, TCP stacks also have various state transitions that heavily affect TCP execution and can trigger semantic bugs. Thus, these approaches fail to cover many state transitions and thus may miss many real bugs. (3) Many of these approaches use common bug sanitizers (such as ASan [2] and MSan [39]) to detect memory bugs; but many bugs in TCP stacks are semantic bugs that are unrelated to memory accesses, and thus common bug sanitizers cannot find these semantic bugs.

In this paper, we propose a novel TCP-stack fuzzing framework named TCP-Fuzz, which consists of three key techniques. First, to generate effective test cases, TCP-Fuzz uses a *dependency-based strategy* that can generate the sequences of syscalls and packets by considering dependencies between them. Specifically, this strategy considers three kinds of dependencies to generate effective test cases, including syscall-syscall, packet-packet and syscall-packet dependencies. For example, a typical packet-packet dependency is that the sequence number of a new packet should be equal to the sum of the sequence number and data length of the previous packet. Second, to effectively cover state transitions, TCP-Fuzz uses a *transition-guided fuzzing approach* that exploits a new coverage metric named *branch transition* as program feedback to replace code coverage. Branch transition is represented as a vector that stores both branch coverage for the current input item (packet or syscall) and the change of branch coverage between the current and previous input items. In this way, branch transition can describe not only states but also state transitions of two adjacent input items. Finally, to detect semantic bugs, TCP-Fuzz uses a *differential checker* that compares the outputs of multiple TCP stacks for the same inputs. Indeed, different TCP stacks should obey many identical semantic rules (most of these rules are defined in RFC documents), and thus they should produce identical or similar outputs for the same inputs. Otherwise, these TCP stacks have implementation inconsistencies, indicating some of them possibly have semantic bugs. This checker is scalable and does not introduce runtime overhead for TCP stacks.

We have implemented TCP-Fuzz with Clang [33] and Packdrill [8]. TCP-Fuzz can detect both memory bugs with existing bug sanitizers and semantic bugs with our differential checker. Overall, we make four main contributions:

- We study TCP stack commits, and find 87% of bug-fixing commits are related to semantic bugs, which cannot be found by existing bug sanitizers. We also reveal the limitations of existing protocol fuzzing in testing TCP stacks.

- To improve fuzzing in testing TCP stacks, we propose three key techniques: (1) a *dependency-based strategy* that considers dependencies between packets and system calls, to generate effective test cases; (2) a *transition-guided fuzzing approach* that uses a new coverage metric named *branch transition* as fuzzing feedback, to improve the coverage of state transitions; (3) a *differential checker* that compares the outputs of multiple TCP stacks for the same inputs, to detect semantic bugs.
- Based on the three key techniques, we design a novel fuzzing framework named TCP-Fuzz, to effectively test TCP stacks. To our knowledge, TCP-Fuzz is the first systematic TCP-stack fuzzing framework to detect both memory and semantic bugs.
- We evaluate TCP-Fuzz on five widely-used user-level and kernel-level TCP stacks (TLDK, F-Stack, mTCP, FreeBSD TCP and Linux TCP), and find 56 real bugs (including 8 memory bugs and 48 semantic bugs). 40 of these bugs have been confirmed by related developers, and 23 bugs have been fixed. Moreover, we also compare TCP-Fuzz to existing fuzzing approaches (AFL-like, Syzkaller-like, Boofuzz, Fuzzotron and AFLNet), and it finds many real bugs missed by these approaches.

The rest of this paper is organized as follows. Section 2 introduces the background and motivation. Section 3 introduces our key techniques of fuzzing TCP stacks. Section 4 introduces TCP-Fuzz. Section 5 shows our evaluation and compares TCP-Fuzz to existing fuzzing tools. Section 6 makes a discussion about fuzzing TCP stacks. Section 7 presents related work, and Section 8 concludes this paper.

2 Background and Motivation

We first introduce TCP stacks, and then we motivate our work by studying TCP stack commits and revealing the limitations of existing protocol fuzzing in testing TCP stacks.

2.1 TCP Stack

The TCP protocol is a classical transport-layer protocol to provide reliable, ordered and error-checked delivery of byte streams via an IP network. In practice, the TCP protocol has different implementations, forming different TCP stacks. Besides classical kernel-level TCP stacks (such as Linux TCP and FreeBSD TCP), many new user-level TCP stacks (such as mTCP, TLDK and F-Stack) have been developed and widely-used to achieve better performance. However, all these TCP stacks has three common features:

F1: Two-dimensional inputs with dependencies. As presented in Figure 1, a TCP stack receives both packets from network drivers and syscalls from applications as inputs, and it outputs the syscalls' results to applications and response packets to network drivers. TCP-related system calls are used

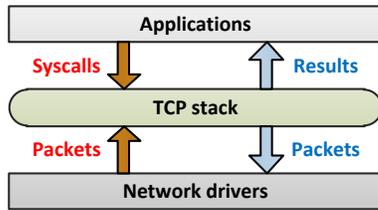


Figure 1: Inputs and outputs of TCP stack.

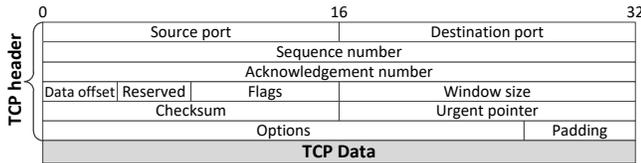


Figure 2: TCP packet format.

to perform fixed network functionalities. For example, the syscall `socket` is used to create an endpoint for communication and it returns a file descriptor of the socket; the syscall `accept` is used to accept a connection on a socket and it returns a new file descriptor of the socket. A TCP packet has a fixed format shown in Figure 2, including a header and data. The TCP header consists of different fields to store the parameters and state of an end-to-end TCP socket.

Packets and syscalls accepted by the TCP stack should have dependencies between each other, otherwise they will be simply neglected by the TCP stack without deep processing. Specifically, there are three kinds of dependencies:

- *Syscall-syscall dependency.* For example, when a connection is passive open, the application must call a series of syscalls including `socket`, `bind`, `listen` and `accept` in order. Otherwise, the application cannot successfully establish the TCP connection.
- *Packet-packet dependency.* For example, after a connection is established, the source port and destination port of each packet should be fixed. Otherwise, the TCP stack identifies the packets to be invalid and thus directly drops them without further processing.
- *Syscall-packet dependency.* For example, the syscall `accept` returns only after the TCP stack receives the last one of the three-way handshake packets.

According to this feature, two requirements should be satisfied when testing TCP stacks. First, it is necessary to generate the sequences of both system calls and packets as input test cases. Second, to make test cases more effective, it is important to consider dependencies between packets and syscalls when generating test cases.

F2: State model. A TCP stack works according to a basic state model defined in the RFC 793 [50] document. Figure 3 shows this basic state model, which has 11 states and 20 state transitions. For a real-world TCP stack, there are often more states and state transitions specific to the TCP stack's implementation.

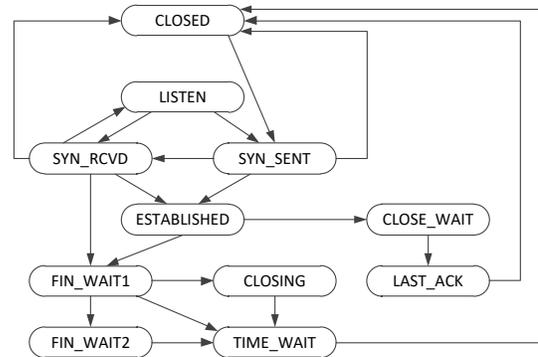


Figure 3: Basic state model of TCP stack.

According to this feature, when testing TCP stacks, it is important to cover both states and state transitions as many as possible. As a state can be reached from different states (for example in Figure 3, `TIME_WAIT` can be reached from `FIN_WAIT1`, `FIN_WAIT2` and `CLOSING`) and there are often more state transitions than states, covering state transitions is actually more important than covering states in testing.

F3: Semantic rules. Each TCP stack works based on some regular semantic rules that stipulate how syscalls and packets should be handled. Most of these semantic rules are explicitly described in RFC documents. For example, the RFC 7323 [49] document describes how to handle the timestamp option in the TCP packet header. An important semantic rule in this RFC document is that once the timestamp option has been successfully negotiated during TCP connection, the TCP stack should accept only packets with non-decreasing timestamps; otherwise the TCP stack should simply drop the packets. However, some semantic rules are not explicitly described in RFC documents. For example, RFC documents defines 32 possible options in the TCP packet header and describes how these options should be handled [58]. But for unknown options, RFC documents do not describe how to handle them. In practice, most TCP stacks simply ignore these options.

According to this feature, when testing TCP stacks, it is important to check these semantic rules and detect related violations. Indeed, these violations are unrelated to problematic memory accesses, and thus we refer them to *semantic bugs*.

2.2 Study of TCP Stack Commits

To understand the proportion of memory bugs and semantic bugs in existing TCP stacks, we select three open-sourced and widely-used TCP stacks, including FreeBSD TCP, mTCP [26] and TLDK [59], to study their commits¹. Among these TCP stacks, FreeBSD TCP is a classical kernel-level TCP stack; mTCP is a well-known user-level TCP stack in academic community; TLDK is a recent user-level TCP stack in industry

¹FreeBSD commits: <https://gitlab.com/FreeBSD/freebsd-src>
mTCP commits: <https://github.com/mtcp-stack/mtcp>
TLDK commits: <https://git.fd.io/tldk/commit/?h=dev-next-socket>

Time	FreeBSD		mTCP		TLDK	
	Memory	Semantic	Memory	Semantic	Memory	Semantic
2017	2	26	2	6	1	11
2018	9	51	0	4	0	4
2019	9	65	1	3	2	5
Total	20	142	3	13	3	20

Table 1: Study results of TCP stack commits.

community and it has been deployed in many telecom systems and network nodes. In our study, we first select the bug-fixing commits from January 2017 to December 2019, resulting in 201 commits; and then we manually read each commit to identify whether it fixes memory bugs or semantic bugs.

Table 1 shows the study results. 87% of bug-fixing commits are related to semantic bugs, namely most of the reported bugs in TCP stacks are semantic bugs. Figure 4 shows an example commit [14] of fixing a semantic bug in FreeBSD TCP stack. The annotation of this commit describes that it fixes a RFC 7323 [49] violation. Specifically, the TCP stack mistakenly accepted the packets with decreasing timestamp values. To fix the bug, this commit adds several checks about the timestamp value to drop invalid packets.

```

FILE: FreeBSD/sys/netinet/tcp_synccache.c
int synccache_expand(...) {
    .....
+   /* RFC 7323 PAWS: if we have a timestamp on this segment and
+    * it is less than ts_recent, drop it.
+   if (sc->sc_flags & SCF_TIMESTAMP && to->to_flags & TOF_TS &&
+       TSTMP_LT(to->to_tsval, sc->sc_tsreflect)) {
+       SCH_UNLOCK(sch);
+       if ((s = tcp_log_addr(s, th, NULL, NULL)) {
+           log(LOG_DEBUG, ...);
+           free(s, M_TCPLOG);
+       }
+       return (-1); /* Do not send RST */
+   }
+   .....
}

```

Figure 4: Example commit of fixing a semantic bug.

In fact, these semantic bugs are introduced for three main reasons. First, because a TCP stack has rich functionalities and complex state model, developers may unintentionally make mistakes about semantic rules when implemented the TCP stack. Second, many semantic rules are used to handle exceptions that infrequently occur in normal execution, and thus the code related to these rules receives insufficient attention in development and testing. Finally, some semantic rules are not explicitly described in RFC documents, and thus developers cannot ensure whether their implemented code obeys these rules. For these reasons, it is important to find semantic bugs in TCP stacks.

2.3 Limitations of Existing Protocol Fuzzing

Fuzzing is an effective technique of runtime testing, and it has shown excellent ability of bug detection in practice. Encouraged by the promising results, many recent approaches perform fuzz testing for the implementations of application-

layer network protocols, such as DTLS/TLS [17, 52, 54, 60], FTP [6, 21, 22, 43] and Modbus [37, 38]. However, we believe that these approaches are limited in testing TCP stacks for three critical reasons:

1) *Fail to generate two-dimensional inputs with dependencies.* Existing fuzzing approaches only generate packets as input test cases, without considering dependencies between inputs. However, as described in *F1* in Section 2.1, TCP stacks receive both syscalls and packets, which have dependencies between each other. If we only generate packets as input test cases, much code about handling syscalls cannot be covered; if we ignore dependencies between system calls and packets, many generated test cases will be meaningless and neglected by TCP stacks without deep processing, which seriously damages fuzzing efficiency. Thus, we need to design a new strategy to generate effective test cases for TCP stacks.

2) *Neglect the coverage of state transitions.* Existing protocol fuzzing approaches use code coverage as program feedback to cover different protocol states. However, as described in *F2* in Section 2.1, besides states, TCP stacks also have many state transitions that heavily affect TCP execution. Moreover, two test cases covering the same states may cover different state transitions. For example in Figure 5, the test case T1 covers the states S1, S2 and S3 in order, and then the test case T2 covers the states S1, S3 and S2 in order. T1 and T2 both cover the states S1, S2 and S3, and thus existing fuzzing approaches identifies T2 to be useless, as it fails to cover new states. But T1 and T2 cover different state transitions, namely T1 covers S1->S2 and S2->S3 while T2 covers S1->S3 and S3->S2. Thus, T2 is useful in covering new state transitions.

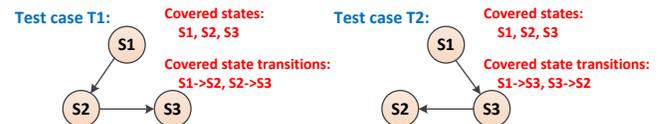


Figure 5: Example of covering states and state transitions.

3) *Lack effective detection of semantic bugs.* Most existing fuzzing approaches use common bug sanitizers (such as ASan [2] and MSan [39]) to detect memory bugs, such as null-pointer dereferences and use-after-free issues. However, as described in Section 2.2, most of the reported bugs in TCP stacks are semantic bugs, which are not caused by problematic memory accesses. Thus, these bug sanitizers cannot detect semantic bugs in TCP stacks.

3 Key Techniques

To solve the limitations of existing fuzzing in testing TCP stacks, we propose three key techniques: a *dependency-based strategy* to generate effective test cases, a *transition-guided fuzzing approach* to improve the coverage of state transitions and a *differential checker* to detect semantic bugs. We introduce these techniques as follows.

3.1 Dependency-Based Strategy

Inspired by existing two-dimensional fuzzing approaches [29, 64] for file systems, we generate *input sequences* containing syscalls and packets as test cases for TCP stacks. Considering that packets and syscalls accepted by TCP stacks have many dependencies with each other, we design a dependency-based strategy to generate more effective test cases for TCP stacks. Given an original input sequence that improves testing coverage, our strategy mutates it to generate new input sequences. As shown in Figure 6, for each item in the original input sequence, our strategy first selects a mutation type and then mutates this item by considering dependencies with the previously handled items.

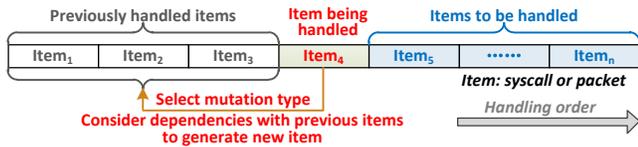


Figure 6: Input sequence mutation.

Mutation-type selection. According to possible operations on a syscall or packet, our strategy provides five available types of mutation (including deletion, addition, replacement and two kinds of changes), as listed in Table 2. Our strategy randomly selects a mutation type to handle each item in the input sequence in order. As a result, different items in the input sequence can be handled with different mutation types.

Item type	Mutation type
Syscall	<i>Deletion</i> : delete this syscall.
	<i>Addition</i> : add a new syscall or packet.
	<i>Replacement</i> : replace this syscall with a new packet.
	<i>Change1</i> : change the parameter of this syscall.
	<i>Change2</i> : change the syscall type with the same parameter.
Packet	<i>Deletion</i> : delete this packet.
	<i>Addition</i> : add a new syscall or packet.
	<i>Replacement</i> : replace this packet with a new syscall.
	<i>Change1</i> : change the TCP header fields of this packet.
	<i>Change2</i> : change the TCP data length of this packet.

Table 2: Available mutation type.

Dependency-based generation. In Table 2, all of the mutation types except deletion generate a new syscall or packet in the input sequence. As described in F1 in Section 2.1, there are three kinds of dependencies between packets and syscalls. If an input sequence violates these dependencies, it is considered to be invalid and can be simply neglected by TCP stacks. Thus, to generate more effective test cases, our strategy considers these dependencies to generate each item in the input sequence. Specifically, when handling an item, our strategy considers the dependencies between this item and the previously handled items. At present, we have implemented 15 dependency rules in Table 3, by referring to RFC documents (packet-packet and syscall-packet dependencies) and syscall-usage conventions (syscall-syscall dependencies).

Kind	Dependency rule
Syscall-syscall	SS1: socket, bind, listen and accept are called in order when a connection is passive open.
	SS2: socket and connect are called in order when a connection is active open.
	SS3: The file descriptor that socket or accept returns is used byfcntl, ioctl, read, write and other syscalls.
	SS4: read and write can be called only after accept or connect is called and returns a success.
	SS5: read and write are never called after close is called.
Packet-packet	PP1: After a connection is established, the source port and destination port of each packet are fixed.
	PP2: The order and control flags of three-way handshake packets and four-way handshake are never changed.
	PP3: The sequence number of a packet is equal to the sum of sequence number and data length of the previous packet.
	PP4: The timestamps of packets are non-decreasing.
	PP5: The echo reply value in timestamp of a packet is equal to the echo value in timestamp of the previous received packet.
Syscall-packet	SP1: accept can be called only after the three-way handshake when a connection is passive open.
	SP2: connect can be called only before the three-way handshake when a connection is active open.
	SP3: Packets can be sent only after accept or connect is called and returns a success.
	SP4: The relative acknowledge number of a packet sent to the stack is no more than total length of data sent by write.
	SP5: After close is called, a packet with the FIN flag should be sent.

Table 3: Implemented dependency rules.

Considering that each TCP stack is implemented according to RFC documents and syscall-usage conventions, we believe that these dependency rules are general to all TCP stacks.

Note that to test whether TCP stacks correctly obey these dependency rules, our strategy also generates some *exceptional input sequences* by deliberately violating packet-packet and syscall-packet dependency rules, with a small probability. Indeed, such input sequences are useful in detecting RFC violations about exception handling.

3.2 Transition-Guided Fuzzing Approach

As described in Section 2.3, code coverage cannot describe state transitions, and thus our fuzzing approach requires a new coverage metric that can effectively describe both states and state transitions.

For a given input sequence, the TCP stack’s state is always changed when handling each item (a syscall or packet) in this sequence. Namely, each such item affects the execution situation of the TCP stack. Thus, after handling each item, the TCP stack can be considered to reach a new state. This state can be described with branch coverage (namely the coverage of code branches), as existing fuzzing approaches do. Accordingly, a state transition can be described as the transition between two covered states due to two adjacent input items, namely the change of branch coverage between these input items. Inspired by this idea, we propose a new coverage metric named *branch transition* to describe both states and state transitions. For a given input sequence, a branch transition is represented as a vector that stores both branch coverage for the current input item and the change of branch coverage between the current and previous input items.

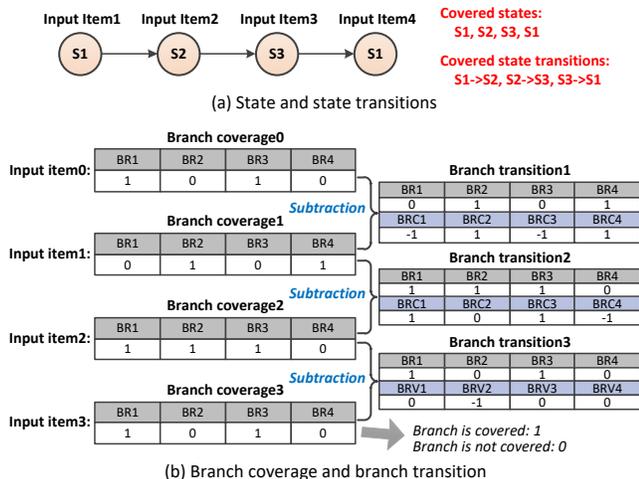


Figure 7: Example of branch transition.

Figure 7 illustrates branch transition using an example. Each state is described using a branch coverage vector, which contains the executed situation (covered or not covered) of each branch in TCP stack code. Then, the state change between the current and previous input items is represented as the subtraction of their branch coverage vectors (*current* – *previous*). Finally, the branch transition of the current input item is obtained as a two-dimensional vector containing its branch coverage vector and the calculated subtraction vector. In Figure 7(a), an input sequence contains four input items which cover the states S1, S2, S3 and S1 in order, and thus it covers three different state transitions S1->S2, S2->S3 and S3->S1. These state transitions are described as three different branch transitions in Figure 7(b). If code coverage is used, *input item3* is identified to be useless, as it covers an old state S1 that is already covered by *input item0*. However, *input item3* actually covers a new state transition S3->S1, which can be successfully described using branch transition.

Our fuzzing approach uses branch transition as program feedback, to effectively cover both states and state transitions. For a given input sequence, if it covers new branch transitions, our fuzzing approach identifies it to be interesting and puts it into the seed corpus for future mutation. Then, our fuzzing approach selects a seed input sequence from the seed corpus and mutates it to generate new input sequences using our dependency-based strategy. We implement most of the fuzzing process by referring to AFL [1].

In fact, besides branch transition, state transition can be also represented as higher-level state change learned by several recent approaches of fuzzing DTLS/TLS protocol implementations [17, 52]. However, the state models learned by these approaches can have mistakes, and thus they still require much manual guidance and validation to ensure correctness. By contrast, branch transition can be automatically and conveniently obtained by collecting runtime information of TCP stacks. Thus, our approach uses branch transition instead of higher-level state change learned by these approaches.

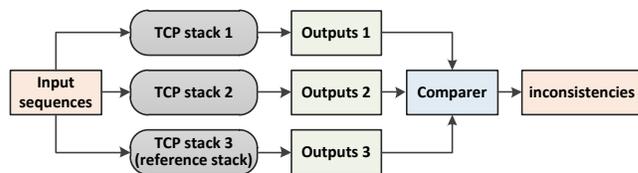


Figure 8: Procedure of our differential checker.

3.3 Differential Checker

To detect semantic bugs, an intuitive solution is to implement semantic checkers by referring to semantic rules in RFC documents. But there are many RFC documents and some semantic rules are even implicit, and thus it is hard to manually implement these checkers. Indeed, different TCP stacks should obey identical semantic rules (many of these rules are defined in RFC documents), and thus they should produce identical or similar outputs for the same inputs. Otherwise, these TCP stacks have implementation inconsistencies, indicating that some of them possibly have semantic bugs.

Based on this idea and inspired by recent approaches of differential testing [11, 12, 65], we design a differential checker for TCP stacks to detect semantic bugs that cause output inconsistencies. As shown in Figure 8, our differential checker provides the same input sequences to multiple TCP stacks, then records their outputs (including return values and parameters of syscalls as well as response packets from TCP stacks), and finally compares these outputs to identify and report inconsistencies. The user can check these inconsistencies to find related semantic bugs.

To improve the efficiency of finding semantic bugs, we suggest using at least one classical and well-tested kernel-level TCP stack (such as Linux TCP or FreeBSD TCP) as a *reference stack* in our differential checker, to test relatively newer TCP stacks. In this case, if our checker reports inconsistencies, it is very likely that one newer TCP stack has semantic bugs.

Our differential checker has three main advantages. First, because different TCP stacks should obey identical semantic rules, the possibility of producing inconsistencies for the same inputs is not large. Thus, the manual work of checking the differences reported by our checker should be much less than that of implementing well-verified checkers of semantic rules. Second, we believe that our checker is also helpful to extracting implicit semantic rules, through identifying and analyzing implementation inconsistencies of multiple TCP stacks. Finally, our checker is scalable and does not introduce runtime overhead for TCP stacks.

At present, our checker records and compares final outputs of TCP stacks, without recording and checking intermediate information (such as window size and packet time) of TCP stacks during packet transmission. Thus, it cannot detect semantic bugs about congestion control and performance. Moreover, our checker detects output inconsistencies between multiple TCP stacks, instead of checking specific RFC doc-

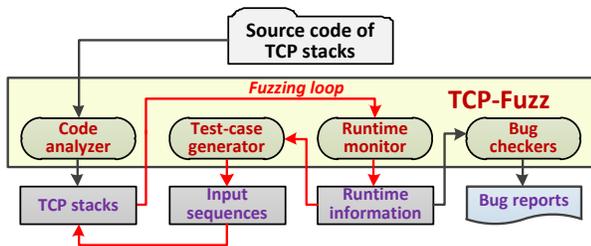


Figure 9: TCP-Fuzz architecture.

uments at runtime. The user needs to manually check RFC documents and analyze the root causes of these inconsistencies, to identify semantic bugs about RFC violations.

4 Framework

Based on the three key techniques in Section 3, we propose a novel fuzzing framework named TCP-Fuzz, to effectively test TCP stacks and detect bugs. We have implemented TCP-Fuzz using Clang 9.0 [13] and Packetdrill [8]. Specifically, we use Clang to perform code instrumentation on TCP stack code, in order to collect covered branches during TCP-stack execution; and we use Packetdrill to send the generated input sequences of syscalls and packets to TCP stacks, and to receive return values and parameters of syscalls as well as response packets from TCP stacks. Overall, TCP-Fuzz consists of four parts:

Code analyzer. It first uses Clang to compile the source code of TCP stacks into LLVM bytecode. Then, it instruments each code branch in the LLVM bytecode. Finally, it compiles the modified LLVM bytecode to generate executable TCP stacks.

Test-case generator. It uses our transition-based fuzzing approach and dependency-based strategy to generate input sequences of syscalls and packets. Each such input sequence is presented as a Packetdrill script, and it is provided to the TCP stacks via Packetdrill. Note that Packetdrill does not support sending some exceptional input sequences that violate dependency rules in Table 3. Thus, we modify Packetdrill by dropping some related checks in its code, to make it support sending such exceptional input sequences.

Runtime monitor. It collects two kinds of runtime information. First, it collects covered branches and calculates branch transitions to provide feedback to our fuzzing approach. Second, it calls Packetdrill interfaces to receive the outputs of each TCP stack, and provides them to our differential checker.

Bug checkers. TCP-Fuzz has three kinds of bug checkers to detect both memory bugs and semantic bugs:

- *Third-party sanitizers.* Existing bug sanitizers (such as ASan [2] and MSan [39]) are used to detect memory bugs by monitoring memory accesses at runtime.
- *Data validator.* We implement this checker to detect semantic bugs leading to incorrect data transfer of TCP stacks, because ensuring data-transfer correctness is a basic property of TCP stacks. Specifically, this checker

performs two kinds of validation: (1) whether the data received by the TCP stack via calling `read` is identical to the data stored in packets sent to the TCP stack; (2) whether the data sent from the TCP stack via calling `write` is identical to the data stored in packets received by the remote end.

- *Differential checker.* This checker is used to compare the outputs of multiple TCP stacks for the same inputs, in order to detect semantic bugs.

Deployment. As shown in Figure 10, TCP-Fuzz is deployed in a server-client mode. In this way, TCP-Fuzz can not only use third-party bug sanitizers and data validator in each TCP stack to detect memory bugs and data-correctness-related bugs, but also use the differential checker in multiple TCP stacks to detect their semantic bugs. The TCP-Fuzz server and clients can be deployed in the same machine and communicate with each other via virtual network controllers; or they can be deployed in different machines and communicate with each other via physical network controllers.

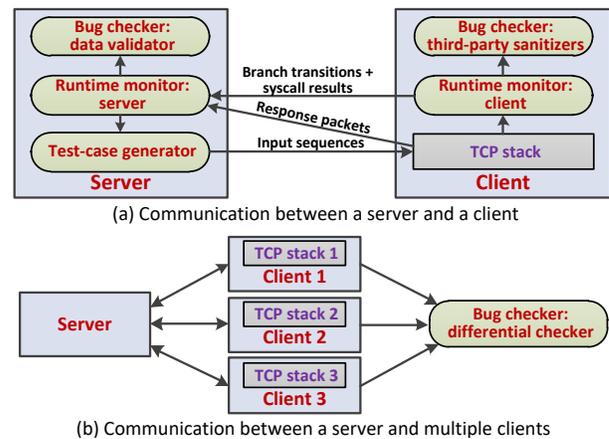


Figure 10: Server-client deployment of TCP-Fuzz.

5 Evaluation

5.1 Experimental Setup

To validate the effectiveness of TCP-Fuzz, we use it to actually test five open-sourced and widely-used TCP stacks, including three user-level ones (TLDK, F-Stack and mTCP) and two kernel-level ones (FreeBSD TCP and Linux TCP). For the three user-level TCP stacks, we test them with the complete fuzzing process of TCP-Fuzz. For the two kernel-level TCP stacks, because they are classical and well-tested, we use them as reference stacks in the differential checker. Moreover, because TCP-Fuzz can only instrument user-level programs at present, we only test the two kernel-level TCP stacks using test cases generated from the user-level TCP stacks, without feedback-driven fuzzing. In the future, we will implement kernel-code instrumentation to support complete fuzzing of kernel-level TCP stacks.

Table 4 shows the basic information about the five tested TCP stacks. Among them, FreeBSD TCP and Linux TCP are two classical kernel-level TCP stacks used in lots of machines; mTCP is a well-known user-level TCP stack in academic community; TLDK and F-Stack are two recent user-level TCP stacks in industry community, and they have been widely deployed in telecom systems and network nodes.

Type	TCP stack	Version	LOC
User-level	TLDK [59]	v2.0	15K
	F-Stack [19]	Commit 8d21adc	25K
	mTCP [26]	Commit 0463aad	18K
Kernel-level	FreeBSD	v12.1	171K
	Linux	v5.6	169K

Table 4: Basic information about tested TCP stacks.

We deploy TCP-Fuzz clients on five regular personal computers, each of which runs a TCP stack to be tested. We deploy TCP-Fuzz server on another personal computer to generate test cases and compare the outputs of these TCP stacks. For each user-level TCP stack, we test it for 48 hours; for each kernel-level TCP stack, we test it by inputting the test cases generated from the three user-level TCP stacks. Besides, we run a third-party sanitizer ASan [2] to detect memory bugs in the user-level TCP stacks.

5.2 Runtime Testing

Table 5 shows the fuzzing results, including covered branches and branch transitions as well as found memory bugs and semantic bugs. Note that TCP-Fuzz does not instrument the two kernel-level TCP stacks, and thus their covered branches and branch transitions are not obtained.

Testing coverage. TCP-Fuzz covers many more branch transitions than branches, indicating that TCP stacks have more state transitions than states during execution. Figure 11 shows the growth of covered branches and branch transitions for the three user-level TCP stacks during fuzzing. Similar to existing fuzzing approaches based on code coverage, TCP-Fuzz covers few new branches during the later tests, but it still covers many new branch transitions during these tests.

Found bugs. TCP-Fuzz finds 56 real bugs in the five tested TCP stacks, including 8 memory bugs and 48 semantic bugs. We reported these bugs to related developers, and 40 of them have been confirmed. We are still waiting for responses for the remaining bugs (for example, the mTCP code in github has not been updated for a long time, and thus we have not received any response to our reported bugs in mTCP). Besides, 23 of the confirmed bugs have been fixed.

Output inconsistencies. TCP-Fuzz reports 15.1K inconsistencies between the five tested TCP stacks, and we analyze their root causes to identify semantic bugs, through our manual review of RFC documents and observation of TCP stack execution. Similar to SQLancer [51] and libFuzzer [32], for inconsistencies that we identify as semantic bugs, we manually

Stack	Testing coverage		Found bugs	
	Branch	Transition	Memory / Semantic	Confirmed / Fixed
TLDK	1.3K	329.4K	2 / 26	28 / 19
F-Stack	7.5K	46.8K	1 / 6	6 / 1
mTCP	1.2K	47.9K	5 / 9	0 / 0
FreeBSD	-	-	0 / 6	5 / 2
Linux	-	-	0 / 1	1 / 1
Total	10.0K	424.1K	8 / 48	40 / 23

Table 5: Results of fuzzing TCP stacks.

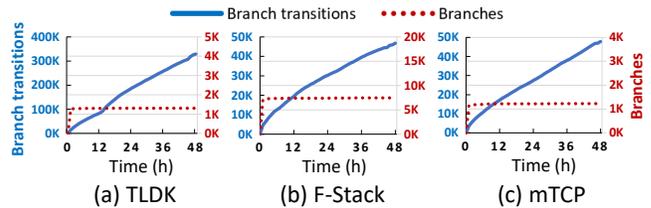


Figure 11: Covered branches and branch transitions.

fix them using the developers' patches or by ourselves, to reduce related inconsistencies. We iteratively repeat this process until inconsistencies never occur, to count unique semantic bugs, resulting in 48 semantic bugs. We observe that many output inconsistencies are repeated, as they are triggered by the same root cause. Only one output inconsistency is considered to be benign. Specifically, when normal packets come after a FIN packet and their sequence numbers are larger than that of the FIN packet, FreeBSD, F-Stack and mTCP drop the FIN packet, while Linux TCP and TLDK reset the connection. As RFC documents do not stipulate how to handle this case, we are not sure which strategies are correct.

Bug-finding process. We also analyze how TCP-Fuzz finds these 56 bugs, and show the results in Table 6. The 8 memory bugs are all found by ASan, 2 semantic bugs are found by the data validator and 46 semantic bugs are found by the differential checker. The results indicate that our differential checker is effective in finding semantic bugs. Besides, we also highlight that 2 semantic bugs found by the data validator are quite dangerous, because they directly cause TCP stacks to send or receive incorrect data, badly damaging data-transfer correctness. Moreover, 28 bugs are found via exceptional input sequences generated by deliberately violating the dependency rules listed in Table 3, while 28 bugs are found via normal input sequences generated by obeying these rules. The results indicate that the TCP stack code about handling exceptional inputs is error-prone in practice. Thus, exception handling in TCP stacks should receive more attention in testing.

Stack	Number of tests	Checker			Input sequence type	
		ASan	Data	Differential	Exceptional	Normal
TLDK	123K	2	1	25	11	17
F-Stack	128K	1	1	5	6	1
mTCP	170K	5	0	9	4	10
FreeBSD	421K	0	0	6	6	0
Linux	421K	0	0	1	1	0
Total	1,263K	8	2	46	28	28

Table 6: Statistics of bug-finding process.

Stack	RFC violation	Syscall issues	Implicit rules
TLDK	15	9	2
F-Stack	5	1	0
mTCP	7	2	0
FreeBSD	5	1	0
Linux	1	0	0
Total	33	13	2

Table 7: Root causes of semantic bugs.

RFC document	791	793	1122	5961	6093	6691	7323	7413
Semantic bug	2	7	1	6	1	1	12	3

Table 8: Distribution of RFC violations.

Root causes of memory bugs. For the 8 found memory bugs, 2 are use-after-free issues, 3 are null-pointer dereferences, 2 are buffer-overflow issues, and 1 is a division-by-zero issue.

Root causes of semantic bugs. For the 48 found semantic bugs, we summarize three root causes in Table 7 and find that:

(1) 33 semantic bugs are RFC violations, and they violate semantic rules explicitly described in the 8 RFC documents shown in Table 8. For example, 6 semantic bugs (3 in TLDK, 1 in mTCP, 1 in F-Stack and 1 in FreeBSD TCP) are RFC 5961 [48] violations. Indeed, the RFC 5961 document is designed to mitigate the influence of blind in-window attacks [36] by changing the range of acceptable sequence numbers in reset packets and acknowledge numbers in normal packets. Thus, these semantic bugs can be exploited by attackers to reset the connection [62] or inject malicious data [7, 10, 45] via blind in-window attacks.

(2) 13 semantic bugs are caused by incorrect results of syscalls. For example, 3 semantic bugs (1 in TLDK, 1 in F-Stack and 1 in FreeBSD TCP) are caused by using an invalid file descriptor obtained from `socket` after `listen` and `accept` are called in order. Indeed, after `listen` and `accept` are called, the previous file descriptor obtained from `socket` becomes invalid, and thus `ioctl`, `read` and `write` should return an error code when using this file descriptor. However, TLDK, F-Stack and FreeBSD TCP return zero to indicate a success in this case, causing semantic bugs.

(3) 2 semantic bugs in TLDK are caused by violating implicit semantic rules. Specifically, RFC documents do not describe how to handle unknown options in the TCP packet header. In our tests, F-Stack, mTCP, FreeBSD TCP and Linux TCP simply ignore these options and accept related packets, but TLDK drops related packets or enters an infinite loop when handling these options.

5.3 Influences of the Found Bugs

We manually review the 56 found bugs to estimate their influences on the reliability and security of TCP stacks. The results are shown in Table 9. We find that 6 semantic bugs about RFC 5961 violations are vulnerable to the MIMT (Man-in-the-middle) attacks; 9 bugs (including 4 memory bugs and 5 semantic bugs) can cause data corruption; 8 bugs (including

Stack	MIMT attack	Corruption	Crash/DoS	Functional error	Inefficiency
TLDK	3	6	4	12	3
F-Stack	1	1	0	3	2
mTCP	1	2	4	3	4
FreeBSD	1	0	0	3	2
Linux	0	0	0	0	1
Total	6	9	8	21	12

Table 9: Reliability and security influence of the found bugs.

4 memory bugs and 4 semantic bugs) can cause crashes or denial of services; 21 semantic bugs can cause functional errors of data communication; and 12 semantic bugs can reduce the efficiency of data communication.

Figure 12 shows three bugs found by TCP-Fuzz, including 1 memory bug and 2 semantic bugs. This figure also shows the related test cases in form of Packetdrill scripts generated by TCP-Fuzz for finding these bugs.

Use-after-free issue in TLDK. In Figure 12(a), the function `rx_fin` free the data of `s->tx.q` by calling `empty_tq`. Then, this data is used by accessing `m->data_len` in the function `txq_rst_nxt_head`, causing a use-after-free issue. Once this bug is triggered, attackers can modify the data of `s->tx.q` to inject malicious data in the TCP connection. To fix this bug, the developer submits a patch to assign zero to `s->tcbs.nd.una_offset` in the function `rx_fin`, in order to avoid accessing the data of `s->tx.q` in the function `txq_rst_nxt_head`.

RFC 7323 violation in FreeBSD TCP. In Figure 12(b), as shown in the annotation of the function `syncache_expand`, if timestamps are not negotiated in the first two packets of three-way handshake, FreeBSD TCP rejects the third packet containing the timestamp option and resets the TCP connection. However, the RFC 7323 document stipulates that the third packet in this case should be normally accepted. Once this bug is triggered, the TCP connection can be abnormally disconnected during three-way handshake, causing a functional error. To fix this bug, the developer submits a patch to modify the problematic code according to the related semantic rule in the RFC 7323 document.

RFC 793 violation in mTCP. In Figure 12(c), if the sequence number of the current packet is smaller than the next expected sequence number, mTCP drops the current packet. However, the RFC 793 document stipulates if the current packet overlaps the range of the expected receive window, this packet should be accepted. Once this bug is triggered, data communication can be inefficient due to abnormally dropping packets. To fix this bug, our preliminary solution is to accept the content of the current packet within the range of expected receive window.

5.4 Comparison to Existing Fuzzing Tools

We perform the comparison in two ways. First, we compare TCP-Fuzz to two classical and widely-used fuzzing approaches, namely AFL [1] and Syzkaller [57]. Considering

the root causes of these inconsistencies requires TCP-specific knowledge that is hard to extract as fixed patterns.

Testing congestion control. Congestion control is an important functionality for TCP stacks, but TCP-Fuzz cannot test it at present, due to ignoring congestion-control-related packet information (such as the length of accepted data for each packet) and code information (such as the variables about congestion window size). In the future, we will collect and check such information by referring to related work [27, 56], to test congestion control implementations of TCP stacks.

Limitations and future works. TCP-Fuzz can be strengthened in some aspects. First, as described in Section 5.1, TCP-Fuzz cannot instrument kernel code in the current implementation, and thus it fails to completely test kernel-level TCP stacks. To solve this limitation, we plan to perform kernel-code instrumentation or tune existing VM-based approaches [25, 55] in TCP-Fuzz. Second, TCP-Fuzz fails to record intermediate information (such as window size and packet time) of TCP stacks during packet transmission, and thus it cannot detect semantic bugs about congestion control and performance. To solve this limitation, we plan to record such intermediate information and implement related checkers to detect these semantic bugs. Third, TCP-Fuzz fails to check concurrent memory accesses, and thus it cannot find concurrency bugs in TCP stacks. To solve this limitation, we plan to introduce existing concurrency-analysis approaches [18, 31] to detect concurrency bugs in TCP stacks. Finally, QUIC [46] is a new and promising transport-layer network protocol proposed, and it is expected to replace TCP in the future. Thus, we also plan to extend TCP-Fuzz to testing QUIC implementations.

7 Related Work

7.1 Network Protocol Fuzzing

Fuzzing is a popular testing technique to detect bugs in software systems. Many fuzzing approaches have been proposed to test the implementations of network protocols.

Some approaches [6, 37, 38, 54, 60] use grammar-based fuzzing. They utilize hard-coded or user-defined grammar specifications to guide test-case generation. These specifications define data structure or field types of packets to be generated. For example, TLS-Attacker [54] is a flexible TLS testing framework for developers to test their TLS implementations by writing Java code or XML-based specifications.

Several recent approaches [17, 43, 52] perform stateful protocol fuzzing. AFLNet [43] can learn basic state models of network protocols to improve seed selection and mutation. Fiterau-Brostean et al. [17] propose a practical tool by extending TLS-Attacker [54], to learn comprehensive state models of multiple DTLS implementations. By comparing these learned state models, the user can infer vulnerabilities in DTLS implementations.

However, these approaches are limited in testing TCP stacks. First, these approaches only generate packets as test cases, but TCP stacks receive both packets and syscalls as inputs, and thus these approaches may miss much code for handling syscalls. Second, these approaches use code coverage as program feedback to cover states, but code coverage cannot effectively describe state transitions in TCP state model. Finally, many of these approaches only use existing bug sanitizers to detect memory bugs, but fail to detect semantic bugs. To solve these limitations, TCP-Fuzz uses a dependency-based strategy to generate effective test cases of packets and syscalls, a transition-guided fuzzing approach to improve the coverage of state transitions, and a differential checker to detect semantic bugs of TCP stacks.

7.2 TCP Stack Checking

Packetdrill [8] is a scripting tool to test the correctness and performance of network stacks. The user can write tcpdump-like scripts to generate and maintain test cases for new feature development and regression testing of network stacks. But writing effective test cases in form of Packetdrill scripts requires a deep understanding of TCP stacks and much manual effort. To solve this problem, TCP-Fuzz automatically generates Packetdrill scripts as test cases according to program feedback and dependencies between packets and syscalls.

Some approaches [24, 34, 40, 41, 53] perform model checking or formal verification of TCP stacks. For example, Lockefeer et al. [34] use μ CRL and LTSmin [35] toolsets to generate state spaces and perform formal verification of TCP extended with the Window Scale Option. Hoque et al. [24] use symbolic execution to precisely simulate program execution with symbolic inputs and explore all possible execution paths, and also use an off-the-shelf model checker to check temporal properties of TCP stacks. But these approaches require much manual effort and TCP-specific knowledge to provide a complete and correct TCP state model, and they are often time-consuming due to high complexity of TCP state transitions.

Some approaches [9, 30] perform static analysis of TCP stack source code. For example, PacketGuardian [9] uses static taint analysis to check the packet handling logic of various network protocol implementations, to detect packet-injection vulnerabilities. However, these approaches often introduce false positives in practice, due to lacking exact runtime information for analysis.

Some approaches [3–5, 66] analyze execution traces of TCP stacks to infer RFC violations. For example, Bishop et al. [4] analyze the execution traces with higher-order logic specifications, to identify differences between multiple network protocols stacks and thus to detect possible RFC violations. However, these approaches require substantial and effective test cases to achieve high testing coverage. To solve this problem, TCP-Fuzz automatically generates effective test cases with fuzzing.

Some approaches [27, 28] perform runtime testing for automated attack discovery of TCP stacks. These approaches strategically generate packets to cover different TCP states, which are tracked according to packet information and pre-defined protocol state machines, without modifying TCP stack code. Different from these approaches, TCP-Fuzz generates both packets and syscalls as test cases, with the guidance of branch transition; TCP-Fuzz does not require pre-defined protocol state machines, but it performs code instrumentation on TCP stacks to collect branch transition.

7.3 Differential Testing

To find semantic bugs, many approaches [11, 12, 23, 42, 47, 61, 65] perform differential testing to identify implementation inconsistencies between multiple programs of the same functionalities. Classfuzz [12] and Classming [11] syntactically mutate Java bytecode files and execute them on different JVM implementations, to identify their inconsistencies. The two approaches both use Markov Chain Monte Carlo (MCMC) sampling to guide mutator selection to improve test-case generation. C2V [65] uses randomized differential testing to detect bugs in code coverage tools (such as gcov and llvm-cov). It randomly generates program code files and compares coverage reports of code coverage tools to identify inconsistencies. Inspired by these approaches, we design a useful differential checker to detect semantic bugs in TCP-stack fuzzing.

8 Conclusion

In this paper, we develop a novel fuzzing framework named TCP-Fuzz, to effectively test TCP stacks and detect bugs. It uses three key techniques: (1) a dependency-based strategy that considers dependencies between packets and system calls, to generate effective test cases; (2) a transition-guided fuzzing approach that uses branch transition as program feedback, to improve the coverage of state transitions; (3) a differential checker that compares the outputs of multiple TCP stacks for the same inputs, to detect semantic bugs. We have evaluated TCP-Fuzz on five widely-used TCP stacks, and find 56 real bugs (including 8 memory bugs and 48 semantic bugs). We also compare TCP-Fuzz to existing fuzzing approaches, and it finds many real bugs missed by these approaches.

In the future, we plan to improve TCP-Fuzz to detect congestion control issues and performance problems, and to apply TCP-Fuzz to other TCP stacks and QUIC implementations.

Acknowledgment

We thank our shepherd, Cristina Nita-Rotaru, and anonymous reviewers for their helpful advice on the paper. We also thank the developers of TCP stacks, who gave useful feedback and advice to us. This work was supported by the Natural Science

Foundation of China under Project 62002195 and the China Postdoctoral Science Foundation under Project 2019T120093. Jia-Ju Bai is the corresponding author.

References

- [1] American Fuzzy Lop. <http://lcamtuf.coredump.cx/afl/>.
- [2] ASan: address sanitizer. <https://github.com/google/sanitizers/wiki/AddressSanitizer>.
- [3] Steve Bishop, Matthew Fairbairn, Hannes Mehnert, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: rigorous test oracle specification and validation for TCP/IP and the Sockets API. *Journal of the ACM*, 66(1):1:1–1:77, 2018.
- [4] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *Proceedings of the ACM SIGCOMM 2005 Conference*, pages 265–276, 2005.
- [5] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *Proceedings of the 33rd International Symposium on Principles of Programming Languages (POPL)*, pages 55–66, 2006.
- [6] Boofuzz: network protocol fuzzing for humans. <https://github.com/jtpereyda/boofuzz>.
- [7] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V. Krishnamurthy, and Lisa M. Marvel. Off-path TCP exploits: global rate limit considered dangerous. In *Proceedings of the 25th USENIX Security Symposium*, pages 209–225, 2016.
- [8] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkupati, Hsiao-Keng Jerry Chu, Andreas Terzis, and Tom Herbert. packetdrill: scriptable network stack testing, from sockets to packets. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, pages 213–218, 2013.
- [9] Qi Alfred Chen, Zhiyun Qian, Yunhan Jack Jia, Yuru Shao, and Zhuoqing Morley Mao. Static detection of packet injection vulnerabilities: a case for identifying attacker-controlled implicit information leaks. In *Proceedings of the 22nd International Conference on Computer and Communications Security (CCS)*, pages 388–400, 2015.

- [10] Weiteng Chen and Zhiyun Qian. Off-path TCP exploit: how wireless routers can jeopardize your secrets. In *Proceedings of the 27th USENIX Security Symposium*, pages 1581–1598, 2018.
- [11] Yuting Chen, Ting Su, and Zhendong Su. Deep differential testing of JVM implementations. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 1257–1268, 2019.
- [12] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th International Conference on Programming Language Design and Implementation (PLDI)*, pages 85–99, 2016.
- [13] Clang: a LLVM-based compiler for C/C++ program. <https://clang.llvm.org/>.
- [14] FreeBSD commit bc35229fad1f: add PAWS check for ACK segments in syncache code. <https://gitlab.com/FreeBSD/freebsd-src/commit/bc35229fad1f>.
- [15] CVE-2019-11478. <https://nvd.nist.gov/vuln/detail/CVE-2019-11478>.
- [16] Aled Edwards and Steve Muir. Experiences implementing a high performance TCP in user-space. *ACM SIGCOMM Computer Communication Review*, 25:196–205, 1995.
- [17] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS implementations using protocol state fuzzing. In *Proceedings of the 29th USENIX Security Symposium*, pages 2523–2540, 2020.
- [18] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*, pages 215–228, 2011.
- [19] F-Stack: high performance network framework based on DPDK. <http://www.f-stack.org>.
- [20] Fuzzotron: a network fuzzer supporting TCP, UDP an multithreading. <https://github.com/denandz/fuzzotron>.
- [21] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: stateful black-box fuzzing of proprietary network protocols. In *Proceedings of the 11th International Conference on Security and Privacy in Communication Systems, Security and Privacy in Communication Networks*, pages 330–347, 2015.
- [22] Serge Gorbunov and Arnold Rosenbloom. Autofuzz: automated network protocol fuzzing framework. *International Journal of Computer Science and Network Security (IJCSNS)*, 10(8):239, 2010.
- [23] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. Dlfuzz: differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 International Symposium on Foundations of Software Engineering (FSE)*, pages 739–743, 2018.
- [24] Endadul Hoque, Omar Chowdhury, Sze Yiu Chau, Cristina Nita-Rotaru, and Ninghui Li. Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs. In *Proceedings of the 47th International Conference on Dependable Systems and Networks (DSN)*, pages 627–638, 2017.
- [25] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Ruzzer: finding kernel race bugs through fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 754–768, 2019.
- [26] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 489–502, 2014.
- [27] Samuel Jero, Md. Endadul Hoque, David R. Choffnes, Alan Mislove, and Cristina Nita-Rotaru. Automated attack discovery in TCP congestion control using a model-guided approach. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.
- [28] Samuel Jero, Hyojeong Lee, and Cristina Nita-Rotaru. Leveraging state information for automated attack discovery in transport protocol implementations. In *Proceedings of the 45th International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2015.
- [29] 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 27th International Symposium on Operating Systems Principles (SOSP)*, pages 147–161, 2019.
- [30] Nupur Kothari, Ratul Mahajan, Todd Millstein, Ramesh Govindan, and Madanlal Musuvathi. Finding protocol manipulation attacks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 26–37, 2011.

- [31] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In *Proceedings of the 27th International Symposium on Operating Systems Principles (SOSP)*, pages 162–180, 2019.
- [32] libFuzzer: a library for coverage-guided fuzz testing. <https://lvm.org/docs/LibFuzzer.html>.
- [33] LLVM compiler infrastructure. <https://lvm.org/>.
- [34] Lars Lockfefer, David M. Williams, and Wan J. Fokkink. Formal specification and verification of TCP extended with the window scale option. *Science of Computer Programming (SCP)*, 118:3–23, 2016.
- [35] LTSmin: model checking and minimization of labelled transition systems. <https://ltsmin.utwente.nl/>.
- [36] Matthew Luckie, Robert Beverly, Tiange Wu, Mark Allman, and kc claffy. Resilience of deployed TCP to blind attacks. In *Proceedings of the 2015 Internet Measurement Conference (IMC)*, pages 13–26, 2015.
- [37] Zhengxiong Luo, Feilong Zuo, Yu Jiang, Jian Gao, Xun Jiao, and Jiaguang Sun. Polar: function code aware fuzz testing of ICS protocol. *ACM Transactions on Embedded Computing Systems*, 18(5s):93:1–93:22, 2019.
- [38] Zhengxiong Luo, Feilong Zuo, Yuheng Shen, Xun Jiao, Wanli Chang, and Yu Jiang. ICS protocol fuzzing: coverage guided packet crack and generation. In *Proceedings of the 57th International Design Automation Conference (DAC)*, pages 1–6, 2020.
- [39] MSan: memory sanitizer. <https://github.com/google/sanitizers/wiki/MemorySanitizer>.
- [40] Sandra L. Murphy and A. Udaya Shankar. Service specification and protocol construction for the transport layer. In *Proceedings of the ACM SIGCOMM 1988 Conference*, pages 88–97, 1988.
- [41] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proceedings of 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 155–168, 2004.
- [42] Yannic Noller, Corina S. Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunke. HyDiff: hybrid differential software analysis. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, pages 1273–1285, 2020.
- [43] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNet: a greybox fuzzer for network protocols. In *Proceedings of the 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465, 2020.
- [44] Lei Qian and Brian E. Carpenter. A flow-based performance analysis of TCP and TCP applications. In *Proceedings of the 18th International Conference on Networks (ICON)*, pages 41–45, 2012.
- [45] Zhiyun Qian, Z. Morley Mao, and Yinglian Xie. Collaborative TCP sequence number inference attack: how to crack sequence number under a second. In *Proceedings of the 19th International Conference on Computer and Communications Security (CCS)*, pages 593–604, 2012.
- [46] QUIC: a multiplexed stream transport over UDP. <https://www.chromium.org/quic>.
- [47] Gaganjeet Singh Reen and Christian Rossow. DPIFuzz: a differential fuzzing framework to detect DPI elusion strategies for QUIC. In *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC)*, pages 332–344, 2020.
- [48] RFC 5961: improving TCP’s robustness to blind in-window attacks. <https://tools.ietf.org/html/rfc5961>.
- [49] RFC 7323: TCP extensions for high performance. <https://tools.ietf.org/html/rfc7323>.
- [50] RFC 793: TCP (Transmission Control Protocol). <https://tools.ietf.org/html/rfc793>.
- [51] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *Proceedings of the 14th International Symposium on Operating Systems Design and Implementation (OSDI)*, pages 667–682, 2020.
- [52] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *Proceedings of the 24th USENIX Security Symposium*, pages 193–206, 2015.
- [53] Mark Anthony Shawn Smith. *Formal verification of TCP and T/TCP*. PhD thesis, Massachusetts Institute of Technology, 1997.
- [54] Juraj Somorovsky. Systematic fuzzing and testing of TLS libraries. In *Proceedings of the 23rd International Conference on Computer and Communications Security (CCS)*, pages 1492–1504, 2016.
- [55] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent Byunghoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamoto: accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *Proceedings of the 29th USENIX Security Symposium*, pages 2541–2557, 2020.

- [56] Wei Sun, Lisong Xu, and Sebastian Elbaum. Scalably testing congestion control algorithms of real-world TCP implementations. In *Proceedings of the 2018 International Conference on Communications (ICC)*, pages 1–7, 2018.
- [57] Syzkaller: an unsupervised coverage-guided kernel fuzzer. <https://github.com/google/syzkaller>.
- [58] Possible TCP options. <https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml>.
- [59] TLDK: Transport Layer Development Kit in network. <https://github.com/FDio/tldk/>.
- [60] Andreas Walz and Axel Sikora. Exploiting dissent: towards fuzzing-based differential black-box testing of TLS implementations. *IEEE Transactions Dependable Secure Computing (TDSC)*, 17(2):278–291, 2020.
- [61] Zhongjie Wang, Shitong Zhu, Yue Cao, Zhiyun Qian, Chengyu Song, Srikanth V. Krishnamurthy, Kevin S. Chan, and Tracy D. Braun. SymTCP: eluding stateful deep packet inspection with automated discrepancy discovery. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS)*, 2020.
- [62] Paul Watson. Slipping in the window: TCP reset attacks. *Technical Whitepaper*, 2004.
- [63] Shinae Woo, Eunyoung Jeong, Shinjo Park, Jongmin Lee, Sunghwan Ihm, and KyoungSoo Park. Comparison of caching strategies in modern cellular backhaul networks. In *Proceeding of the 11th International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 319–332, 2013.
- [64] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 818–834, 2019.
- [65] Yibiao Yang, Yuming Zhou, Hao Sun, Zhendong Su, Zhiqiang Zuo, Lei Xu, and Baowen Xu. Hunting for bugs in code coverage tools via randomized differential testing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 488–499, 2019.
- [66] Yanyan Zhuang, Eleni Gessiou, Steven Portzer, Fraida Fund, Monzur Muhammad, Ivan Beschastnikh, and Justin Cappos. Netcheck: network diagnoses from black-box traces. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 115–128, 2014.

MLEE: Effective Detection of Memory Leaks on Early-Exit Paths in OS Kernels

Wenwen Wang
University of Georgia

Abstract

Memory leaks in operating system (OS) kernels can cause critical performance and security issues. However, it is quite challenging to detect memory leaks due to the inherent complexity and large-scale code base of real-world OS kernels. In this work, inspired by the observation that software bugs are often hidden in rarely-tested program paths, we focus on detecting memory leaks on *early-exit (E-E) paths* in OS kernels. To this end, we conduct a systematic study of memory management operations involved on E-E paths in OS kernels. Based on the findings, we design a novel leak detector for OS kernels: MLEE, which intelligently discovers memory leaks on E-E paths by *cross-checking* the presence of memory deallocations on different E-E paths and normal paths. MLEE successfully reports 120 *new* memory leak bugs in the Linux kernel. It is the first time these memory leaks are uncovered by a leak detector for OS kernels.

1 Introduction

Memory leaks are a common class of memory management bugs and wide spread in many critical software systems. The accumulation of leaked memory objects can eventually exhaust the limited memory resource, leading to a significant influence on system response time and throughput [6, 18, 37], as well as shortened battery life on mobile devices [43]. Besides, memory leaks have been exploited to launch security attacks, e.g., CVE-2019-12379 [39] and CVE-2019-8980 [40].

Over the past decades, significant efforts have been devoted to detecting memory leaks [4, 7, 14, 16, 22, 27, 31, 32, 44–46]. However, most of them are designed based on the structures and features of user-level applications and thus cannot be applied directly to operating system (OS) kernels. Compared to user applications, OS kernels are much more complicated in terms of program logic and semantic [33], as they need to handle various exceptional statuses, respond to arbitrary interrupts from peripheral devices, perform security checks on untrusted data sources, and etc. Besides, given the gigantic

```
1 /* mm/mempool.c */
2 int mempool_resize(mempool_t *pool, int new_min_nr) {
3     ...
4     spin_lock_irqsave(&pool->lock, flags);
5     if (new_min_nr <= pool->min_nr) {
6         spin_unlock_irqrestore(&pool->lock, flags);
7         kfree(new_elements); // A memory deallocation.
8         return 0;
9     }
10    ...
11    return 0;
12 }
```

Figure 1: An example of E-E path in the Linux kernel.

code base of a typical OS kernel, e.g., 25 million source code lines of the Linux kernel with hundreds of new lines added per hour, it is extremely challenging to complete the leak detection for the entire kernel within an acceptable time.

In this paper, we focus on detecting memory leaks on *early-exit paths* (or E-E paths for short) in OS kernels. This is inspired by the observation that software bugs often lurk in rarely-tested program paths [7, 44]. In general, an E-E path is designed to exit from a kernel routine as early as possible. But, before the routine is exited from the E-E path, some extra work usually needs to be completed, e.g., deallocating a memory object, as shown by the example in Figure 1. Hence, if a memory deallocation is required but *missed* on the E-E path, it constitutes a memory leak. Though recent work also attempts to find software bugs related to E-E paths [19, 21, 35], their schemes are limited to a specific type of E-E paths, i.e., error handlers, and thus cannot cover a broad range of buggy E-E paths. Our study reveals that, besides error handlers, E-E paths in OS kernels are used in many *previously-unknown* scenarios, which inherently render existing approaches ineffective to detect memory leaks on general E-E paths.

There are two typical reasons why memory leaks are particularly common on E-E paths. First, the major purpose of an E-E path is to exit from a kernel routine as soon as possible when a special system status is encountered. With this in

mind, it is fairly easy for a developer to compose an E-E path with some required work missed on the path, especially when the developer does not have a comprehensive knowledge of the required work. Second, in practice, many E-E paths are added to the kernel code base not during the development but after the deployment due to various reasons, e.g., correcting the processing logic, achieving better performance/reliability, or enhancing the security. This inevitably leads the E-E paths to the lack of complete and thorough testing, which has been demonstrated by previous work [17].

To understand how E-E paths are used in OS kernels and how memory management operations are involved on E-E paths, we systematically investigate a substantially large portion, i.e., around 1 million lines, of the source code of two popular OS kernels, Linux [2] and FreeBSD [1]. The study uncovers some interesting findings on common usage scenarios of E-E paths and general principles of memory deallocations on E-E paths. In particular, we observe that the presence *in-consistencies* of memory deallocations on different E-E paths and normal paths usually indicate potential memory leaks. Inspired by this observation, we design MLEE to intelligently detect memory leaks on E-E paths through *cross-checking* the presence of memory deallocations on different paths. MLEE also employs several novel static analysis techniques to balance the analysis efficiency and detection accuracy.

We have implemented MLEE based on LLVM [24], which is a popular compiler infrastructure. We evaluate MLEE by applying it to the Linux kernel. MLEE is able to complete the analysis for the entire Linux kernel in around half an hour. This shows the analysis efficiency and scalability of MLEE. By manually analyzing the report produced by MLEE, we finally confirm 120 new leaks. It is the first time these bugs are uncovered by a leak detector. Most of them have been acknowledged by Linux developers and fixed using our patches. For the others, we are working closely with the kernel developers to finalize the patches.

In summary, this paper makes the following contributions:

- We conduct a comprehensive study of E-E paths in OS kernels and involved memory management operations. The study discovers some interesting and inspiring findings. To the best of our knowledge, this is the first systematic study of memory management operations on E-E paths.
- We design MLEE, which employs effective and scalable static program analysis techniques to identify E-E paths and analyze memory deallocations for memory leak detection. We believe the proposed analysis techniques will also benefit similar bug-detection systems.
- We find 120 new memory leak bugs in the Linux kernel with the help of MLEE. It is the first time that these bugs are reported by a leak detector. Most of these bugs have been acknowledged by Linux maintainers and fixed by our patches. This demonstrates the capability of MLEE to detect memory leaks in a real-world OS kernel.

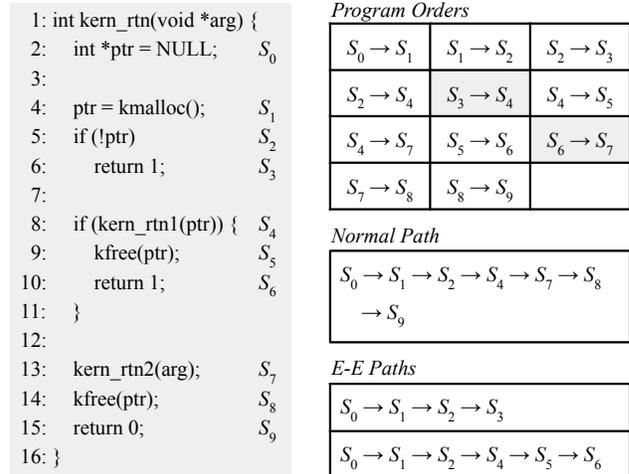


Figure 2: An example of the definition of E-E paths.

2 Background and Motivation

This section gives a formal definition of E-E paths and elaborates common usage scenarios of E-E paths in OS kernels.

2.1 What is E-E Path?

Intuitively, an E-E path intends to exit from a kernel routine *earlier* than normal program paths, by skipping some of the program code in the routine without execution. Hence, an E-E path generally has less program statements than a normal program path. We next give a definition of E-E paths.

A Formal Definition. We use a two-tuple to denote a kernel routine: $R = \langle S, O \rangle$, where S is a set of statements: S_0, \dots, S_n , and O is a set of program orders between two statements: $S_i \rightarrow S_j, 0 \leq i, j \leq n$. Each statement represents a basic and concrete operation, e.g., assigning a value to a kernel variable or invoking a callee routine. The program orders specify how the statements can be executed according to the orders in which they show up in the source code. For example, $S_i \rightarrow S_j$ means that S_j can be executed after the execution of S_i because, for example, S_j shows up in the source code immediately after S_i . In this paper, we call S_j a *successor* statement of S_i . It is worth noting that a return statement may also have a successor statement under O , as it may show up in the middle of a routine. This is the key point of E-E paths.

Given $R = \langle S, O \rangle$, a *program path* of R is an ordered list of statements: $P = S_0 \rightarrow S_1 \rightarrow \dots \rightarrow S_m$, where

- $S_i \in S, 0 \leq i \leq m$;
- $S_i \rightarrow S_{i+1} \in O, 0 \leq i \leq m - 1$;
- S_m is a return statement.

We enforce a return statement at the end of a program path to facilitate the following definition of E-E path. In case there

```

1 /* ipc/mqueue.c */
2 static void mqueue_evict_inode(struct inode *inode)
3 {
4     ...
5     clear_inode(inode);
6     if (S_ISDIR(inode->i_mode))
7         return;
8     /* Evict the inode from the message queue. */
9     ...
10 }

```

Figure 3: An E-E path for irrelevant kernel state bypassing.

is no return statement in a kernel routine, we can append an implicit return statement at the end of the routine source code. P is an E-E path if the following condition satisfies:

$$\exists S_n \in \mathcal{S}, \text{ s.t. } S_m \rightarrow S_n \in \mathcal{O} \quad (1)$$

The rationale behind this definition is that the return statement of an E-E path P is *not* the last statement of the routine, i.e., it has a successor statement under \mathcal{O} . Hence, if the routine exits from P , the successor statement and the following statements will not be executed, leading to an *early* exit.

An Example. We next use the example in Figure 2 to explain the above definition. The left side of the figure shows the source code of the kernel routine, which has ten statements with two if branches: S_2 and S_4 , and three returns: S_3 , S_6 , and S_9 . The top right side of the figure shows the program orders. Note that the two return statements S_3 and S_6 have successor statements, as highlighted in the figure. This routine has three possible program paths and two of them are E-E paths. For example, $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3$ is an E-E path, as S_3 has a successor statement under the program orders: $S_3 \rightarrow S_4$.

2.2 Usage Scenarios of E-E Paths

Though E-E paths are often used to handle unexpected errors, e.g., memory allocation failures and invalid function arguments, our study on popular OS kernels, i.e., Linux and FreeBSD, reveals that E-E paths are also composed in other usage scenarios. We next describe each scenario in detail.

Error/Exception Handling. In general, an OS kernel needs to deal with various exceptional system statuses and unexpected program errors. For example, when a permission check fails, the kernel should not continue to perform the following privileged and safety-critical operations. Otherwise, potential security risks will be raised. Therefore, E-E paths are extensively used in OS kernels to terminate unsafe and invalid execution when an error/exception happens.

Typically, an E-E path in this usage scenario returns an *error code* to indicate that an error is triggered. This is also the reason why existing approaches leverage error codes to detect error handlers [12, 19]. However, our study shows that this is not always true. Specifically, we find that 52% (out of

```

1 /* fs/fat/misc.c */
2 void fat_time_unix2fat(struct msdos_sb_info *sbi, ...)
3 {
4     ...
5     if (tm.tm_year < 1980 - 1900) {
6         *time = 0;
7         *date = cpu_to_le16((0 << 9) | (1 << 5) | 1);
8         return;
9     }
10    if (tm.tm_year > 2107 - 1900) {
11        *time = cpu_to_le16((23 << 11) | (59 << 5) | 29);
12        *date = cpu_to_le16((127 << 9) | (12 << 5) | 31);
13        return;
14    }
15    ...
16 }

```

Figure 4: Two E-E paths for kernel functionality extension.

5910) and 25% (out of 1700) E-E paths in Linux and FreeBSD respectively do not return any error code even if they handle errors/exceptions. In fact, whether an E-E path returns an error code primarily depends on what error/exception is handled by the E-E path. Thus, it is insufficient to detect E-E paths merely relying on whether an error code is returned.

Irrelevant Kernel State Bypassing. In this scenario, the E-E paths are used to bypass specific kernel states, as it is not necessary for the following program code to process these states. The specific kernel states are often represented in the form of flags, modes, sizes, capacities, properties, etc. Figure 3 shows such an example. Here, if `inode` indicates not a regular file but a directory, it is unnecessary to evict it from the message queue. Thus, an E-E path is created to skip such inodes.

It is worth pointing out that this usage scenario is *not* same as the previous scenario. The major difference is that the E-E paths in this scenario handle *legal* and *valid* kernel states, while the E-E paths in the previous scenario tackle *unexpected* kernel errors/exceptions. That is, the E-E paths in this scenario may show up in a routine even if no error/exception needs to be handled. Besides, if an E-E path in this scenario returns a value, it usually returns the same value as normal paths. In contrast, an E-E path in the previous scenario typically returns a value (if any) different from those of normal paths.

Kernel Functionality Extending. An E-E path in this usage scenario extends existing program logic in a kernel routine to incorporate additional functionalities. Figure 4 shows an example. Here, the routine converts a UNIX date to a (time, date) pair in the FAT file system. However, FAT only supports years between 1980 and 2107. Hence, two E-E paths are created for years outside of this range. E-E paths in this scenario often contain extra statements to extend the functionalities. A representative example of E-E paths in this scenario is *fast paths*, which aim to accelerate the processing of some special cases. Previous research shows that fast paths are very likely to introduce bugs [17], which aligns with our observations.

2.3 Memory Management on E-E Paths

On the surface, memory management has no relationship with E-E paths. However, our experience shows that more than 60% kernel routines that have memory management operations contain at least one E-E path. So a further study is necessary to understand how memory management operations are involved on E-E paths. We next report our observations.

Observation 1. *If a memory object is allocated in a kernel routine, it usually needs to be deallocated on the following E-E paths in this routine.* In principle, after a memory object is allocated, it should be used in the following computations. However, in reality, it is possible that an E-E path is reached before the object is actually used. Hence, the object has to be cleaned up on the E-E path, otherwise a memory leak may be introduced. This observation implies that memory deallocation is an important component of the cleanup work on E-E paths. In fact, 58% and 41% E-E paths investigated by our study in Linux and FreeBSD respectively contain at least one memory deallocation. On the other side, if an allocated memory object is used before an E-E path is taken, it is probably not required to deallocate this object on the E-E path. But, we indeed witness some cases where memory objects are deallocated on the following E-E paths even if they are used before. This demonstrates the close correlation between memory deallocations and E-E paths.

Observation 2. *If a memory object is deallocated on the normal paths of a kernel routine, it usually also needs to be deallocated on the E-E paths in this routine.* This observation shows the consistency between normal paths and E-E paths on deallocating memory objects. If an object is deallocated on normal paths, it implies that the live range of this memory object is limited to the current kernel routine, so it should be deallocated on E-E paths as well. We find that 50% and 48% of kernel routines in Linux and FreeBSD respectively have similar deallocations on E-E paths and normal paths. This provides a strong evidence for the analyses in MLEE, as we will see in Section 4. In particular, MLEE pays special attention to memory deallocations on normal paths and employs effective and scalable static analyses to check whether these deallocations are also required on E-E paths. This allows MLEE to detect required but missed deallocations on E-E paths.

Observation 3. *If a memory object is deallocated on an E-E path in a kernel routine, it usually needs to be deallocated on the following E-E paths with same/similar return values (if any) in the routine.* If two E-E paths in a routine return same/similar values, it usually indicates that these two paths are constructed for same/similar purposes, e.g., in the same usage scenario. Particularly, if a memory object is deallocated on an E-E path, it means the caller routine is irresponsible to deallocate this object under the return value of the E-E path, and thus, this object should be deallocated on the following E-E paths with same/similar return values. Therefore, MLEE

creates scalable and precise static analyses to detect memory deallocations present on some E-E paths but missed on others.

Observation 4. *If multiple memory objects are allocated in a kernel routine, all objects allocated before an allocation failure usually need to be deallocated on the E-E path corresponding to this allocation failure.* One of the common reasons for multiple memory allocations in the same routine is to allocate *semantically-correlated* memory objects. For example, the allocation of a memory object with a subfield pointing to a child object is often realized by firstly allocating the object itself and then allocating the child object. Thus, if the allocation for the child object fails, the previously-allocated object has to be deallocated. This observation drives MLEE to thoroughly inspect kernel routines with multiple memory allocations to detect memory leaks. If one of the memory objects is deallocated on an E-E path but not on a following E-E path, it probably indicates a memory leak.

3 Issues and Challenges of MLEE

The design of MLEE is inspired by the key insight that an *inconsistency* between the presence of a memory deallocation on an E-E path and its presence on a normal path or another E-E path in the same kernel routine very likely indicates a potential memory leak. Therefore, it is possible to detect memory leaks by *cross-checking* memory deallocations on different E-E paths and normal paths in the same routine. Though the idea is intuitive, there are several technical challenges.

How to identify early-exit paths? Given the vast amount and the complicated logic of the source code in an OS kernel, e.g., more than 25 million lines in Linux, it is quite challenging to precisely identify E-E paths in a wide range of kernel modules with significant semantic diversities. Besides, the various usage scenarios of E-E paths, as described in Section 2.2, may lead E-E paths to exhibit different characteristics at the source code level. A simple solution for this issue is to enumerate all possible program paths of a kernel routine and exhaustively examine each of them to see whether it satisfies the definition of E-E paths in Section 2.1. However, this solution is obviously impractical due to the poor scalability caused by the well-known path explosion problem.

To address this challenge, we further investigate the structures and semantics of numerous E-E paths, in particular, with detecting memory leaks in mind. We find that the key for MLEE to detect memory leaks on an E-E path is to find out the crucial features of this E-E path that distinguish it from normal paths and other E-E paths in the same routine. Furthermore, such features are often reflected in a specific portion of the E-E path, i.e., the last several statements at the end of the E-E path. Recall the example in Figure 2. Compared to S_0 , S_1 , and S_2 , the statement S_3 is more representative of the first E-E path, because S_3 is manifested *uniquely* on this E-E path. In a similar way, S_5 and S_6 are more unique for the second

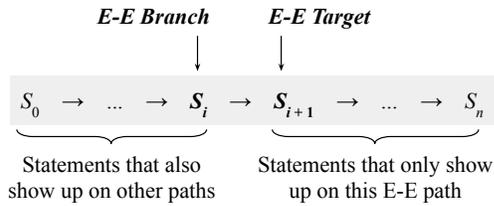


Figure 5: E-E branch and E-E target of an E-E path.

E-E path, as they only show up on this E-E path and thus can be used to distinguish this E-E path from other paths.

In addition, we find out that such statements often follow a conditional branch statement, e.g., an if statement. In this paper, we call such a conditional branch as an *E-E branch*, which is essential to differentiate an E-E path from normal paths and other E-E paths. Similarly, we call the target of the E-E branch that is included by the E-E path as an *E-E target*, which initiates the following statements after the E-E branch on the E-E path. Figure 5 shows the basic structure of an E-E path. It is worth noting that an E-E branch typically show up on both E-E paths and normal paths. Also, it is possible that an E-E path encompasses more than one E-E branch. Here, we primarily consider the *last* E-E branch on the E-E path because it is this one that leads to the unique statements.

In summary, to detect memory leaks on E-E paths, MLEE first collects E-E branches in a kernel routine and then utilizes them as anchor points of following analyses. This allows MLEE to compose precise and scalable static analyses to identify E-E paths and avoid the path explosion issue.

How to analyze memory deallocations? After the E-E branches in a kernel routine are recognized, MLEE moves forward to analyze memory deallocations in this routine to identify the inconsistencies between their presence on different E-E and normal paths. A simple analysis scheme is to examine each deallocation in the routine and check whether it shows up on an E-E path, in particular, after the E-E branch. If not, MLEE then reports a memory leak on this E-E path. Though this scheme sounds reasonable, it may produce plenty of false positive cases and require significant manual efforts to screen them. The major cause of false positives is that the deallocation may not be required on a specific E-E path. For instance, the memory object to be deallocated may be used, e.g., as the return value, on the E-E path and thus should not be deallocated.

To address this issue, MLEE firstly checks whether a missed memory deallocation is required to show up on an E-E path. This is realized in three steps. First, MLEE ensures that the deallocated object is live on the E-E path via a classical context-sensitive and flow-sensitive liveness analysis. Second, MLEE guarantees the validity of the deallocated object on the E-E path, i.e., the object is successfully allocated and remains allocated. Finally, MLEE confirms that the deallocated object

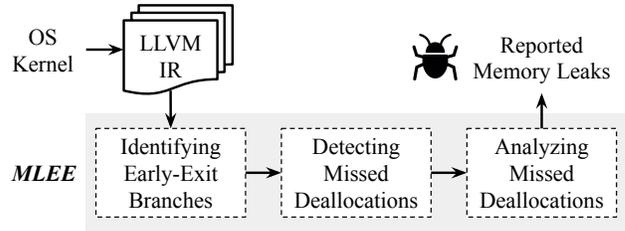


Figure 6: The work flow of MLEE.

is not used on the E-E path, particularly after the E-E branch (see Section 4 for more details). This way, MLEE can infer whether the deallocation is required on the E-E path or not.

4 MLEE System Design

In this section, we firstly present an overview of how MLEE works and then describe the technical details of MLEE.

Figure 6 illustrates the high-level work flow of MLEE. Essentially, MLEE takes as the input the LLVM IR of the target OS kernel and reports potential memory leaks on E-E paths in each kernel routine. We choose to start from LLVM IR because LLVM has plenty of static analysis passes in place, such as control/data-flow and alias analysis. Besides, LLVM IR has quite comprehensive debugging information (generated with the “-g” option), which can map the IR back to the corresponding source code. This allows MLEE to report the source code locations of the detected memory leaks to facilitate further manual inspection and bug fixing. Even though the implementation of MLEE leverages the LLVM infrastructure to reduce engineering effort, it is worth pointing out that the static analyses used by MLEE are *not* provided by LLVM. These analyses are a primary contribution of our work.

For each kernel routine, MLEE firstly analyzes all conditional branches in this routine to identify E-E branches. Next, MLEE gathers memory deallocations in this routine. For each pair of an E-E branch and a memory deallocation, MLEE then checks whether the memory deallocation is *missed* on the E-E paths associated with the E-E branch, i.e., the memory object freed by the deallocation operation is not deallocated after the E-E branch. MLEE focuses the analysis particularly on the statements after the E-E branch, including the E-E target and its following statements, because they are unique and thus more representative of the E-E paths. This also allows MLEE to delimit the analysis scope and scale up the analysis. Besides, it is possible that a memory object is deallocated implicitly after the E-E branch, e.g., in a callee routine. Hence, MLEE creates an inter-procedural analysis to cover all statements in callee routines invoked after the E-E branch.

If the result of the above step is yes, it means the memory deallocation is not present on the associated E-E paths. MLEE then further infers whether the missed deallocation should show up on the E-E path, i.e., whether the correspond-

ing memory object should be deallocated after the E-E branch. To this end, MLEE gathers the liveness, validity, and usage information of the memory object after the E-E branch to guide the analysis. Also, MLEE leverages additional information, e.g., the relative locations of the memory deallocation and the E-E branch in the source code of the kernel routine, to assist the necessity inference, according to the observations in Section 2.3. If the analysis concludes that the memory object needs to be deallocated after the E-E branch, MLEE will report a memory leak bug along with the source code locations of the E-E branch and the missed memory deallocation.

4.1 Identifying E-E Branches

Typically, a conditional branch comes with two branch targets, which are taken when the specified condition is satisfied or not, respectively. The major difference of an E-E branch, compared to a regular conditional branch, is that one of the branch targets is an E-E target, which only shows up on E-E paths, and thus leads to the last several statements on an E-E path. Hence, MLEE determines whether a conditional branch is an E-E branch based on the following two conditions:

- **Condition-1:** The program paths starting from one target always reach to a return statement that has a successor statement under the program orders of the routine.
- **Condition-2:** The program paths starting from another target always have a possibility to reach to a return statement that has no successor statement under the program orders of the routine.

A conditional branch is identified as an E-E branch if its two branch targets satisfy the above two conditions. Here, it is not hard to understand **Condition-1**, because one branch target of an E-E branch should always lead to E-E paths. In fact, the branch target that satisfies **Condition-1** is the E-E target of the E-E branch. The purpose of **Condition-2** is to emphasize the possibility of an E-E branch to show up on normal paths. In essence, an E-E branch divides the following execution space of the current routine into two *disjoint* sets. One of the sets comprises executions only ending with early exits, while another set includes at least one execution ending with normal exit. Therefore, **Condition-2** is crucial to distinguish an E-E branch from a conditional branch both of whose branch targets merely lead to E-E paths.

According to these two conditions, MLEE creates an effective static analysis to identify E-E branches by carefully checking branch targets of each conditional branch. Algorithm 1 shows the details of the static analysis. Overall, for each conditional branch in the kernel routine, MLEE firstly collects its two branch targets, i.e., the two first statements executed immediately after the conditional branch is taken and not taken, respectively, and then checks whether they satisfy the two conditions. If yes, MLEE then identifies this conditional branch as an E-E branch and places it into the set

Algorithm 1: Identification of Early-Exit Branches

Input: R - a kernel routine
Output: $EEBSet$ - the set of early-exit branches in R

```

1  $EEBSet \leftarrow \emptyset$ ;
2  $CBSet \leftarrow \text{Collect\_Conditional\_Branches}(R)$ ;
3 for  $CB \in CBSet$  do
4    $BTSet \leftarrow \text{Collect\_Branch\_Targets}(CB)$ ;
5   for  $T_1 \in BTSet$  do
6      $RetSet_1 \leftarrow \text{Collect\_Reachable\_Returns}(T_1)$ ;
7      $Flag \leftarrow TRUE$ ;
8     for  $Ret \in RetSet_1$  do
9       if  $Ret$  has no successor statement then
10        |  $Flag \leftarrow FALSE$ ;
11        | break;
12      end
13    end
14    if  $Flag \neq TRUE$  then
15      | continue;
16    end
17    for  $T_2 \in BTSet \setminus \{T_1\}$  do
18       $RetSet_2 \leftarrow \text{Collect\_Reachable\_Returns}(T_2)$ ;
19      for  $Ret \in RetSet_2$  do
20        if  $Ret$  has no successor statement then
21          |  $EEBSet \leftarrow EEBSet \cup \{CB\}$ ;
22          | break;
23        end
24      end
25    end
26  end
27 end
28 return  $EEBSet$ ;

```

of identified E-E branches. Otherwise, MLEE continues to check next conditional branch.

To determine if a branch target satisfies one of the two conditions, MLEE traverses the control-flow graph (CFG) of the kernel routine to gather all return statements that are reachable from the branch target. A potential issue in this traversing process is how to handle loop structures. Since MLEE mainly intends to visit all reachable nodes starting from the branch target on the CFG, MLEE iterates a loop as many times as necessary until all of its nodes are visited. This allows MLEE to exhaustively capture all return statements in a loop, which actually are quite common in OS kernels. Once the reachable return statements are collected for the branch target, MLEE next checks each of them to confirm whether it has a successor statement under the program orders of the routine. Given that the static analysis works at the LLVM IR level, MLEE needs to resolve the source location of a return statement. To this end, MLEE leverages the debugging information embedded into the LLVM IR to establish the mapping between the LLVM IR and the corresponding source code [41]. This way, MLEE can determine whether the branch target satisfies one of the two conditions mentioned above.

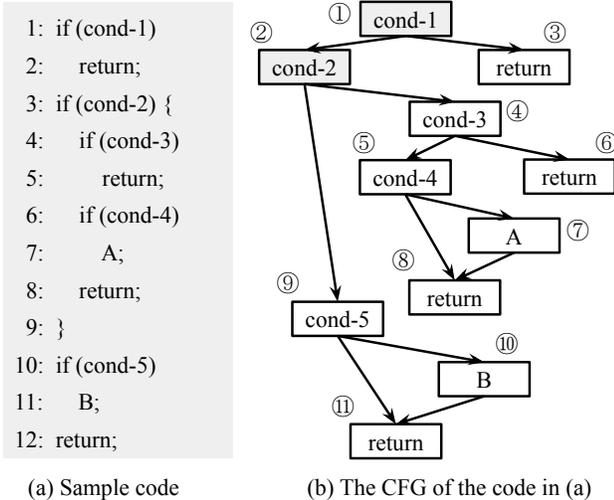


Figure 7: Conditional branch analysis for E-E path detection.

Now, let us use the example in Figure 7 to understand how MLEE identifies E-E branches. The left side of the figure presents the source code and the right side shows the corresponding CFG. From the figure, we can see that there are five conditional branches and four return statements in this example. Besides, each return statement has a successor statement under the program orders except the one at line 12.

MLEE consecutively analyzes each conditional branch to determine whether it is an E-E branch. Take the conditional branch of `cond-1` at line 1 as an example. The *true* target of this conditional branch can only reach to the return statement at line 2, i.e., ① → ③ in the CFG. On the other hand, the *false* target of this conditional branch may reach to the return statement at line 12, e.g., through the program path ① → ② → ⑨ → ⑪ in the CFG. As a result, MLEE identifies this conditional branch as an E-E branch. In a similar way, the conditional branch of `cond-2` is also identified as an E-E branch by MLEE. However, the conditional branches of `cond-3`, `cond-4`, and `cond-5` are not recognized as E-E branches. This is because MLEE figures out that their branch targets do not satisfy the above two conditions after the analysis. For example, the branch targets of the conditional branch of `cond-3` reach to two different return statements at line 5 and line 8, respectively. But both of these two return statements have a successor statement under the program orders. Hence, the conditional branch of `cond-3` is not considered as an E-E branch by MLEE.

4.2 Detecting Missed Memory Deallocation

Given an identified E-E branch *EEB*, the next task of MLEE is to detect memory deallocations that are potentially missed on E-E paths associated with *EEB*.

To this end, MLEE first collects memory deallocations in the same kernel routine as *EEB*. Next, for each deallocation

D, which is suppose to deallocate a memory object *M*, MLEE checks whether *M* is always deallocated before *EEB*. MLEE achieves this by traversing the CFG to search for another deallocation *D'* that also deallocates *M*. More importantly, *D'* dominates *EEB* in the CFG, which means *D'* is on all program paths from the entry of the routine to *EEB* and executed before *EEB*. In other words, if there exists such a *D'*, it implies that *M* has already been deallocated before *EEB* and therefore, *D* should not be considered as a missed deallocation on the E-E paths of *EEB*. Otherwise, MLEE continues to check whether *M* is deallocated by the program statements after the E-E target on the E-E paths. If not, MLEE treats *D* as a missed deallocation of *EEB* and further checks whether it is necessary on the E-E paths of *EEB*.

It is possible that *M* is deallocated in a callee routine of the current kernel routine. Thus, MLEE devises a path-sensitive and context-sensitive inter-procedural analysis to achieve the above goal. Specifically, MLEE analyzes every backward reachable statement starting from the return statement on the E-E paths corresponding to *EEB*. In case the statement is a call statement, MLEE further includes the statements in the callee routine into the analysis.

MLEE pays special attention to each memory deallocation statement during the analysis, because MLEE needs to validate whether it is used to deallocate *M*. A memory object is often accessed through *pointers*, which hold the address of the object. Given that an object can be accessed through different pointers, it is *incomplete* to determine whether two objects freed by two memory deallocations are same or not by simply checking whether the two deallocations use the same pointer. To solve this issue, MLEE leverages the alias analysis in LLVM to discover the potential alias relationship between two pointers. This allows MLEE to differentiate memory deallocations for different objects.

4.3 Analyzing Missed Memory Deallocation

Once a missed memory deallocation *D^M* is identified for an E-E branch *EEB*, the final step of MLEE is to check whether *D^M* is necessary on the E-E paths corresponding to *EEB*, i.e., the memory object *M* deallocated by *D^M* should also be deallocated on the E-E paths.

MLEE firstly composes an effective static analysis to validate the liveness and validity of *M* at the point of *EEB*, as it may cause unexpected program errors to deallocate an out-of-scope memory object. Specifically, MLEE relies on the analyses in LLVM to determine whether *EEB* is covered by the liveness range of *M*. Regarding the validity, MLEE needs to verify the possibility for *M* to be in an “allocated” state when *EEB* is reached. This is because *EEB* may be on a program path that does not allocate *M*. Hence, MLEE firstly conducts a backward slicing on *M* starting from *D^M* to find out the statement that successfully allocates *M*. Then MLEE performs a reachability analysis from the allocation statement

to *EEB* by traversing the CFG.

MLEE needs to take care of two special cases during the above analysis process: 1) No allocation statement is found for *M* because, for example, *M* is allocated in another kernel routine. To solve this issue, MLEE treats the entry point of the current routine as a *nominal* allocation point of *M*. 2) Multiple allocations are found for *M*, due to, for example, different allocation mechanisms adopted by the kernel to allocate *M*. MLEE deals with this case soundly by collecting all allocation statements that can reach *EEB*.

In addition to checking the liveness and validity of *M*, MLEE also needs to ensure that *M* is not used by the statements after *EEB* on the related E-E paths. To this end, MLEE employs a forward slicing on *M* starting from the E-E target of the E-E branch to confirm that *M* is not used by the following statements. The analysis stops when the return statement on the E-E paths of *EEB* is reached. Some common usage examples include but not limited to passing *M* to a callee routine as an argument, accessing a memory location inside of *M*, calculating effective memory addresses based on the start address of *M*, and etc. If *M* is indeed used, MLEE will skip the following analysis on D^M and *EEB*.

Finally, MLEE infers the necessity of D^M for *EEB*. To achieve this, MLEE creates a set of *checking rules* mainly based on the observations described in Section 2.3. The rules check various factors of *EEB* and D^M to heuristically determine whether D^M is required on E-E paths corresponding to *EEB*. To give an example, one of the rules checks the order of D^M and *EEB* in which they appear in the source code. This is inspired by the observation that if D^M shows up earlier than *EEB* in the source code, it typically implies that D^M is executed before *EEB* and therefore, it is very likely that D^M is not necessary on the corresponding E-E paths. In contrast, if the order is reversed, it is probably essential for the corresponding E-E paths to deallocate *M*. The output of each rule is either 1 or 0, meaning the deallocation is required or not on the E-E paths, respectively. MLEE assigns an empirical weight in (0, 1) for each rule and calculates the final result using the following formula:

$$f(EEB, D^M) = \sum_i w_i R_i(EEB, D^M) \quad (2)$$

where w_i is the weight of the *i*th rule: $\sum_i w_i = 1$, and $R_i(EEB, D^M)$ is the output of the *i*th rule for *EEB* and D^M . If the final result exceeds a predefined threshold 0.5, MLEE determines that D^M is required for *EEB* and reports a memory leak with the detailed information of *EEB* and D^M .

5 Implementation

We have implemented MLEE as an LLVM (version 8.0.0) tool composing of multiple analysis passes. This section reports the issues we encountered during the implementation of MLEE as well as the solutions we adopted to address them.

Compiling Linux to LLVM IR. Currently, LLVM is not fully compatible with the Linux kernel. To solve this issue, we compose a Python script to automatically extract and adapt the compilation commands to compile the kernel source code to LLVM IR. We simply skip a source file if LLVM cannot compile it. To include as many kernel modules as possible, we use the “*allyes*” option to configure the kernel. At last, our compilation covers around 11K kernel modules.

Global Call Graph. MLEE conducts the path-sensitive and context-sensitive inter-procedural analyses on a global call graph. To build the graph, MLEE incrementally compose a child call graph for each kernel module when it is loaded and parsed by LLVM [42], and eventually combines them together to obtain the global call graph. Note that MLEE does not need to link all kernel modules together to produce a single LLVM IR file and thus avoids potential linking errors in this process. MLEE also employs a type-based analysis to identify all possible call targets for an indirect call [29, 36].

Alias Analysis. MLEE relies on the alias analysis in LLVM to analyze the alias relationship between two memory pointers. This is achieved by querying the LLVM alias analysis framework with the two memory pointers along with the size of the memory locations. There are four possible relationship outcomes for such a query: *MustAlias*, *PartialAlias*, *MayAlias*, and *NoAlias*. For accuracy consideration, MLEE considers two pointers are alias pointers only when *MustAlias* or *PartialAlias* is returned by the alias analysis framework.

Handling Goto Statements. Some Linux routines use goto statements to realize sophisticated control logic. This entails difficulties for MLEE to detect E-E paths, as the return statement of an E-E path implemented using a goto statement may be placed at the end of a kernel routine. To deal with this issue, MLEE creates a source transformation tool to remove goto statements by replacing them with the target statements they jump to. Note that MLEE only replaces *forward* goto statements because backward goto statements are mostly used to implement loops. Fortunately, the Linux kernel does not use indirect goto statements [11], which take as input label variables and can complicate the transformation.

6 Experimental Study

This section evaluates MLEE by applying it to the Linux kernel (version 5.0, which was the most recent version when we finished the implementation of MLEE). Due to time limitations, MLEE is not evaluated with other OS kernels, e.g., FreeBSD, and non-kernel applications. This is a direction for future work. The evaluation presented in this section would like to answer two research questions about MLEE: i) *Effectiveness*: can MLEE discover new memory leaks in a real-

Table 1: Detection statistics of MLEE. “KEE”: kernel routines that have E-E paths. “KMD”: kernel routines that have memory deallocations. “KEM”: kernel routines that have both E-E paths and memory deallocations. “MMD”: E-E branches that are reported by MLEE to have missed memory deallocations.

Total	Kernel Routine			Mem Dealloc	E-E Branch	
	KEE	KMD	KEM		Total	MMD
887877	121829	14540	7685	20751	297451	126
	13.7%	1.6%	0.9%			0.04%

world OS kernel? ii) *Efficiency*: can MLEE complete the analysis of an entire OS kernel in an acceptable time?

Table 1 shows the detection statistics. As shown in the table, MLEE finds out that 0.9% kernel routines comprise of both E-E paths and memory deallocation operations. After the analysis, MLEE reports that 126 suspicious E-E branches have missed memory deallocations. These E-E branches span across various kernel modules and account for 0.04% of the total E-E branches identified by MLEE. With further manual analysis and investigation, we can finally confirm memory leak bugs on the corresponding E-E paths.

Finding New Bugs. It takes around 3~5 minutes for a researcher to manually investigate one suspicious E-E branch. Finally, we confirm 120 memory leaks related to 103 buggy E-E branches. Note that one buggy E-E branch may correspond to multiple missed deallocations. All of these memory leaks are *new* bugs. This demonstrates the capability of MLEE to discover unknown memory leaks in a real-world OS kernel.

Table 2 shows the details of the found memory leaks. We have reported these bugs along with the patches to Linux developers. During the communication with the developers, we were surprised that the developers responded to our bug reports promptly, e.g., in half an hour or even less, and actively worked with us to revise and update the patches. Given the developers’ heavy maintenance workload, this once again shows the high priority and severity of fixing memory leaks in OS kernels. In summary, at the time of the paper submission,

- 89 bugs (74.2%) have been fixed in the latest version of the Linux kernel using our patches;
- 16 bugs (13.3%) have been fixed in the latest version of the Linux kernel using others’ patches;
- 15 bugs (12.5%) have been confirmed and we are working with the kernel developers to finalize the patches.

Table 2 also shows that a vast majority of memory leaks are detected in the driver (60%), sound (17.5%), and file system (16.7%) directories. This means memory leaks are a general type of software bugs across various kernel modules. We find that 74% missed deallocations are also found on other E-E paths. This shows that E-E paths in the same kernel routine often exhibit similar behaviors in memory management

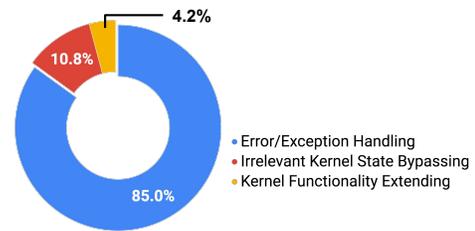


Figure 8: The usage-scenario distribution of the E-E paths on which the memory leaks are detected by MLEE.

```

1 /* drivers/net/ethernet/mellanox/mlx4/en_rx.c */
2 int mlx4_en_config_rss_steer(struct mlx4_en_priv *priv)
3 {
4     ...
5     err = mlx4_qp_alloc(mdev->dev, priv->base_qp, ...);
6     if (err) {
7         en_err(priv, "Failed to allocate RSS ...");
8         goto rss_err; // rss_map->indir_qp is leaked
9     } // on this early-exit path.
10    ...
11    kfree(rss_map->indir_qp);
12    rss_map->indir_qp = NULL;
13    rss_err:
14    ...
15    return err;
16 }

```

Figure 9: A real memory leak in Linux detected by MLEE.

operations. On the other side, this implies memory deallocations on normal paths may also be required on E-E paths.

An interesting observation on the E-E paths of the E-E branches in Table 2 is that these buggy E-E paths are used in not only the scenario of error/exception handling, but also two other scenarios described in Section 2.2. Figure 8 shows the detailed distribution of the scenarios. This demonstrates that MLEE can detect buggy E-E paths in different usage scenarios. However, existing bug-detection schemes that focus only on error handlers may fail to detect these memory leaks.

To understand when and how the found memory leaks are introduced into the kernel code base, we further investigate the code revision histories of the kernel routines in which the memory leaks are found. We find that many memory leaks were actually brought in at the first time when the corresponding E-E paths were added into the kernel source code. This was also acknowledged by some Linux maintainers. Taking the Bug#100 in Table 2 as an example, when we submitted it to the Linux patch mailing list, a kernel maintainer quickly commented our bug report and said that “*this bug exists till its first commit for v2.6.39.*” One possible reason behind this phenomenon is that the author of an E-E path may have a partial and incomplete understanding of the work that are necessary on the E-E path. In addition, this also reveals that many E-E paths in OS kernels are error-prone due to the lack of extensive testing when they are committed to the kernels.

Table 2: New memory leaks in the Linux kernel discovered by MLEE. “O”: the bug has been fixed in the latest version using our patch. “L”: the bug has been fixed in the latest version using others’ patch. “C”: the bug has been confirmed.

	Kernel Routine	E-E Branch	Missed Deallocation		Kernel Routine	E-E Branch	Missed Deallocation	
1	block/...:bio...prep	if (ret...)	kfree(buf)	O	61	drivers/...:i24...it	if (res...)	kfree(opt...) O
2	drivers/...:cm...	if (*ppos...)	kfree(buf)	O	62	drivers/...:e...am	if (e1...)	kfree(tx_old) O
3	drivers/...:acpi...ble	if (!ac...)	kfree(entry)	O	63	drivers/...:e...am	if (e1...)	kfree(rx_old) O
4	drivers/...:fs_open	if (!to)	kfree(vcc)	O	64	drivers/...:ath...te	if (!sta...)	kfree(arsta...) O
5	drivers/...:f...en	if (DO...)	kfree(vcc)	O	65	drivers/...:pr...en	if (ai->...)	kfree(file...) O
6	drivers/...:f...en	if (!tc)	kfree(vcc)	O	66	drivers/...:pr...en	if (down...)	kfree(file...) O
7	drivers/...:read...refs	if (xen...)	kfree(req)	O	67	drivers/...:lbs...ate	if (lbs...)	kfree(cmd) C
8	drivers/...:read...refs	if (xen...)	kfree(ind...)	O	68	drivers/...:mw...d	if (mwi...)	kfree(hos...) L
9	drivers/...:read...refs	if (xen...)	kfree(seg...)	O	69	drivers/...:mw...d	if (!skb)	kfree(hos...) L
10	drivers/...:devfreq...	if (err...)	kfree(dev...)	L	70	drivers/...:mw...d	if (err)	kfree(hos...) L
11	drivers/...:ti_...	if (of_...)	kfree(rsv...)	O	71	drivers/...:fr...pvc	if (*get...)	kfree(pvc) C
12	drivers/...:omap...	if (de...)	kfree(c)	O	72	drivers/...:ya...tl	if (ym...)	kfree(ym) C
13	drivers/...:dr...	if (WA...)	kfree(pl...)	C	73	fs/...:btrfs...mod	if (IS...)	kfree(ref) O
14	drivers/...:md...	if (p_v...)	kfree(dsi...)	C	74	fs/...:btrfs...mod	if (be...)	kfree(ref) O
15	drivers/...:md...	if (!dsi...)	kfree(dsi...)	C	75	fs/...:btrfs...mod	if (be...)	kfree(ra) O
16	drivers/...:md...	if (md...)	kfree(dsi...)	C	76	fs/...:btrfs...mod	if (exi...)	kfree(ref) O
17	drivers/...:c...it	if (pl...)	kfree(pk...)	O	77	fs/...:btrfs...mod	if (act...)	kfree(ref) O
18	drivers/...:i...ed	if (de...)	kfree(de...)	O	78	fs/...:parse...ket	if (*new...)	kmem...(...) O
19	drivers/...:fault...te	if (co...)	kfree(data)	O	79	fs/...:ecryp...ging	if (!ecr...)	kfree(ecr...) O
20	drivers/...:fault...te	if (un...)	kfree(data)	O	80	fs/...:ker...file	if (byt...)	vfree(*buf) L
21	drivers/...:fault...d	if (un...)	kfree(data)	O	81	fs/...:jffs2...block	if (jffs2...)	kfree(su...) O
22	drivers/...:mlx4...fs	if (!tu...)	kfree(tun...)	O	82	fs/...:__br...se	if (!ctx)	kmem...(...) O
23	drivers/...:mlx4...fs	if (ib_...)	kfree(tun...)	O	83	fs/...:_nfs...copy	if (hand...)	kfree(res...) O
24	drivers/...:srp...in	if (!po...)	kfree(addr)	L	84	fs/...:_nfs42...py	if (pro...)	kfree(res...) O
25	drivers/...:led...set	if (trig...)	kfree(event)	O	85	fs/...:nfs4...tion	if (nfs4...)	__free...(...) O
26	drivers/...:led...set	if (dev...)	kfree(event)	O	86	fs/...:nfs4...tion	if (nfs4...)	kfree(loc...) O
27	drivers/...:ra...r	if (rs...)	kfree(rs)	O	87	fs/...:om...map	if (blo...)	kfree(sbi...) C
28	drivers/...:dvb...ty	if (!dv...)	kfree(dv...)	O	88	fs/...:re...mount	if (sb_...)	kfree(new...) C
29	drivers/...:su...bs	if (!us...)	kfree(ca...)	O	89	fs/...:re...mount	if (sb_u...)	kfree(new...) C
30	drivers/...:cx...re	if (ret...)	vfree(p_...)	O	90	fs/...:re...mount	if (!sb...)	kfree(new...) C
31	drivers/...:cx...re	if (ret...)	vfree(p_...)	O	91	fs/...:__ub...ac	if (err)	kfree(hmac) O
32	drivers/...:cx...re	if (i2c...)	kfree(buf)	O	92	fs/...:read_znode	if (ubifs...)	kfree(idx) O
33	drivers/...:dib...on	if (rx0...)	kfree(rx)	O	93	kernel/...:p...t	if (ret...)	kfree(path) L
34	drivers/...:dib...on	if (rx0...)	kfree(tx)	O	94	kernel/...:tr...te	if (!p...)	kfree(par...) O
35	drivers/...:dib...on	if (rx1...)	kfree(rx)	O	95	kernel/...:tr...te	if (!pid...)	kfree(par...) O
36	drivers/...:dib...on	if (rx1...)	kfree(tx)	O	96	lib/...:test...init	if (__te...)	kfree(test...) O
37	drivers/...:h...ch	if (saa...)	kfree(he...)	O	97	net/...:com...ace	if (WA...)	vfree(ent...) O
38	drivers/...:f...n	if (ctx...)	kfree(ctx)	O	98	net/...:eth...stats	if (ret < 0)	vfree(data) L
39	drivers/...:vp...up	if (ma...)	kfree(buf...)	O	99	net/...:bpf...filter	if (!new...)	kfree(addr) C
40	drivers/...:vp...up	if (sub...)	kfree(buf...)	O	100	sound/...:iso...it	if (WA...)	kfree(b->...) O
41	drivers/...:vp...up	if (ti...)	kfree(buf...)	O	101	sound/...:sn...dec	if (snd_...)	kfree(cod...) O
42	drivers/...:sm...ne	if (sm...)	kfree(zo...)	O	102	sound/...:snd...ed	if (w->r...)	kfree(w) O
43	drivers/...:na...t	if (ch...)	vfree(buf)	O	103	sound/...:snd...ed	if (w->p...)	kfree(w) O
44	drivers/...:spi...it	if (spi...)	kfree(dw...)	O	104	sound/...:snd...ed	if (w->clk)	kfree(w) O
45	drivers/...:o...q	if (oct...)	vfree(oct...)	O	105	sound/...:soc...te	if (se==...)	kfree(priv...) L
46	drivers/...:bl...te	if (bit...)	kvfree(t)	O	106	sound/...:soc...te	if (se==...)	kfree(name) L
47	drivers/...:ix...32	if (ix...)	kfree(mask)	O	107	sound/...:soc...te	if (se==...)	kfree(dval...) L

Table 2: (Continued)

48	drivers/...:ix...32	if (ix...)	kfree(input)	O	108	sound/...:soc...te	if (se==...)	kfree(dtex...)	L
49	drivers/...:ix...32	if (ix...)	kfree(jump)	O	109	sound/...:soc...te	if (strnle...)	kfree(priv...)	L
50	drivers/...:ml...er	if (ml...)	kfree(rss...)	O	110	sound/...:soc...te	if (strnle...)	kfree(name)	L
51	drivers/...:m...be	if (st...)	kfree(mg...)	O	111	sound/...:soc...te	if (strnle...)	kfree(dval...)	L
52	drivers/...:q...start	if (qe...)	kfree(p_...)	O	112	sound/...:soc...te	if (strnle...)	kfree(dtex...)	L
53	drivers/...:q...start	if (qe...)	vfree(p_...)	O	113	sound/...:sou...it	if (index...)	kfree(s)	O
54	drivers/...:q...info	if (rc)	kfree(cd...)	C	114	sound/...:hif...init	if (hif...)	kfree(rt)	O
55	drivers/...:cx...nd	if (!time)	kfree(voi...)	O	115	sound/...:hif...init	if (hif...)	kfree(buffer)	O
56	drivers/...:k...dr	if (kal...)	kfree(us...)	O	116	sound/...:hif...init	if (snd_...)	kfree(buffer)	O
57	drivers/...:k...dr	if (kal...)	kfree(us...)	O	117	sound/...:par...unit	if (!kctl)	kfree(nam...)	O
58	drivers/...:l...be	if (reg...)	kfree(buf)	O	118	sound/...:add...ctl	if (err < 0)	kfree(elem)	C
59	drivers/...:l...be	if (lan...)	kfree(buf)	O	119	sound/...:add...ctl	if (err < 0)	kfree(elem)	C
60	drivers/...:i24...init	if (ss...)	kfree(op...)	O	120	sound/...:snd...c3	if (!pd)	kfree(fp->...)	O

Comparing with Existing Leak Detectors. A further study of the memory leaks reported by MLEE shows that many of the leaks actually cannot be detected using existing static detection approaches due to their fundamental limitation. We use the Bug#50 in Table 2 as an example to explain this limitation. Figure 9 shows the source code of this bug. Here, the memory object `rss_map->indir_qp` is not deallocated when `mlx4_qp_alloc()` fails. MLEE successfully reports this memory leak by figuring out that the memory deallocation at line 11 is missed on the E-E path of the E-E branch at line 6.

However, most of existing static detectors are established based on a basic assumption that a memory object should be deallocated at the end of the kernel routine, in which it is allocated, if it does *not* escape from this routine. This assumption itself has no problem, but it cannot cover memory leaks involving *escaped* objects, which are fairly common in OS kernels. Obviously, in this example, `rss_map->indir_qp` is still alive after the routine is returned. Hence, it is extremely hard for existing detectors to reason about whether `rss_map->indir_qp` should be deallocated or not for the buggy E-E path.

False Negatives. For a bug detection tool like MLEE, it is important to study false negatives of the detection results. However, given the large code base of the Linux kernel, it is hard to study false negatives of the detection results of MLEE at the scale of the entire kernel, because of the lack of the ground truth. Nevertheless, during the implementation and evaluation of MLEE, we conducted several false negative experiments to understand the completeness of the static analysis techniques developed in MLEE. For example, we manually examined many kernel routines to verify whether MLEE can identify all E-E paths in these routines, especially when the E-E paths are used in different usage scenarios. In addition, we intentionally modified several randomly selected source files in the kernel to introduce some known memory leak bugs by commenting out memory deallocations

on related E-E paths and checked whether MLEE is able to detect them. Our experimental results showed that MLEE can successfully identify all E-E paths and report the injected memory leaks without any false negative. This demonstrates, to some extent, the completeness of the analysis techniques in MLEE on detecting memory leaks on E-E paths in Linux.

False Positives. As a static leak detector, MLEE inevitably suffers from false positives. Among the reported 126 suspicious E-E branches, 23 (18%) are confirmed having no memory leaks. Generally, a false positive is reported by MLEE due to two categories of reasons. First, the missed memory deallocation is required but invoked after the E-E path is completed, i.e., the current kernel routine is returned. Second, the missed memory deallocation is not required on the E-E path due to various corner cases not handled by MLEE.

We summarize the detailed reasons in each category and the associated percentages in Table 3. The most accountable reason is #5, where the target E-E path is composed for a different semantic purpose compared to the path(s) on which the missed deallocation is found. For example, a memory deallocation may be necessary on E-E paths that handle exceptional kernel statuses but unnecessary for E-E paths that handle expected statuses. To filter out these false positives, we can extend MLEE to capture the semantic differences between paths in the same routine. For other false positive reasons, e.g., #2, we need to develop scalable techniques to continue the tracking of the path after the E-E path is completed.

Analysis Efficiency. The evaluation platform is equipped with an Quad-Core Intel Xeon E5-1620 v4 CPU at 3.50 GHz and 32 GB main memory. The OS is Ubuntu 16.04 with Linux-4.4. The platform was exclusively occupied by MLEE during the analysis process. It took around half an hour for MLEE to complete the analysis of the Linux kernel. Given that the Linux kernel contains 25M lines of source code, we believe this analysis performance is acceptable.

Table 3: Detailed reasons of false positives reported by MLEE and the percentage of each reason.

Required	1	The memory object is deallocated by another kernel thread after the E-E path.	8%
	2	The memory object is deallocated by the same kernel thread in a callback routine after the E-E path.	18%
Not Required	3	The missed deallocation itself is redundant as the allocated object can be reused.	8%
	4	The E-E path does not satisfy a specific condition.	13%
	5	The E-E path is not used for the same semantic purpose as the path(s) of the missed deallocation.	53%

7 Related Work

Static Leak Detection. Static detection approaches analyze programs to discover memory leaks without running the programs. Clouseau [15] identifies program statements inconsistent with an ownership model as memory leaks. Hackett and Rugina [13] propose a region-based shape analysis to detect memory leaks. Xie et al. [44] represents computations as boolean constraints for memory leak detection. Orlovich et al. [31] disprove the presence of memory leaks through a backward data-flow analysis. FastCheck [7] reduces memory leak detection to a reachability problem over a guarded graph. LeakChecker [46] identifies memory leaks through a common code pattern of memory leaks. SMOKE [9] develops the use-flow graph to detect leaks in two stages.

MLEE is also a static memory leak detector. But, MLEE is inspired by the observations and findings from our study of memory management on E-E paths in OS kernels. This enables MLEE to invent effective and scalable static analyses to detect memory leaks on E-E paths. The detection results show that MLEE is a practical leak detector. In contrast, it is quite difficult to directly apply most of existing static mechanisms to OS kernels, due to the extreme complexity of the kernel code base and the widespread use of E-E paths in various scenarios. Although Xie et al. [44] applied the proposed leak detector to the Linux kernel, it took around one day to complete the analysis and, among the reported 123 leaks, only 2 were actually confirmed and fixed. Moreover, the code base of the Linux kernel has expanded significantly since then.

Dynamic Leak Detection. Dynamic detectors identify memory leaks by running the programs. Purify [14] adapts garbage collection techniques to detect memory leaks in C/C++ programs. SWAT [8] and Sleight [4] identify a *stale* object as a memory leak if it has not been accessed for a long time. Cork [22] uses a dynamic heap-summarization technique to detect memory leaks. Rayside et al. [32] leverage object ownership profiling to find instances of memory management anti-patterns in object-oriented programs. Xu et al. [45] propose a heap-tracking technique to detect leaks in Java programs. Hound [30] organizes the heap layout to facilitate the staleness tracking of heap objects for leak detection. Li et al. [26] develop a dynamic technique to validate and categorize memory leak warnings reported by static detectors. GC assertions [3] allow developers to diagnose leaks through a system interface. Maxwell et al. [28] apply a graph grammar mining approach

to detect leaks. Sniper [23] utilizes hardware performance monitoring units to track object staleness for leak detection. Lee et al. [25] train a machine learning model during the software testing stage and apply the model to the production stage for leak detection. MemInsight [20] performs a lifetime analysis on objects to detect leaks in JavaScript programs.

Compared to static detectors, dynamic approaches suffer from the coverage issue, i.e., only the exercised paths are examined. How to generate sufficient test inputs to cover rarely-executed paths, e.g., E-E paths, is an interesting but hard problem. Also, dynamic detectors deployed in production runs may incur performance overhead. MLEE may utilize dynamic techniques to further enhance the detection accuracy.

Others. Some research work aims to automatically fix or tolerate memory leaks. LeakSurvivor [34] and Melt [5] swap out leaked objects to disks to free up memory resources. Leak-Fix [10] checks each allocation and inserts a deallocation if necessary. BLeak [38] provides a leak debugging framework for web applications. Generally, MLEE can collaborate with them to provide integrated solutions for memory leaks.

8 Conclusion

Memory leaks can lead to critical performance and security issues, especially in OS kernels. Inspired by the observation that memory leaks often lurk in rarely-tested program paths, e.g., E-E paths, we firstly conduct a comprehensive and in-depth study on memory management operations involved on E-E paths in OS kernels. With the findings derived from the study, we then design MLEE, which aims to intelligently detect memory leaks on E-E paths. MLEE employs novel static analysis techniques to automatically identify E-E paths and infer whether a missed memory deallocation is required on an E-E path. The static analyses in MLEE are effective and scalable. With the help of MLEE, we discover 120 new memory leaks in the Linux kernel and most of these memory leaks have been fixed using our patches.

Acknowledgments

We are very grateful to our shepherd, Eric Schkufza, and the anonymous reviewers for their valuable feedback and comments. This work is supported in part by a faculty startup funding of the University of Georgia.

References

- [1] The freebsd project. <https://www.freebsd.org>.
- [2] The linux kernel archive. <https://www.kernel.org>.
- [3] Edward E. Aftandilian and Samuel Z. Guyer. Gc assertions: Using the garbage collector to check heap properties. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 235–244, New York, NY, USA, 2009. ACM.
- [4] Michael D. Bond and Kathryn S. McKinley. Bell: Bit-encoding online memory leak detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 61–72, New York, NY, USA, 2006. ACM.
- [5] Michael D. Bond and Kathryn S. McKinley. Tolerating memory leaks. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 109–126, New York, NY, USA, 2008. ACM.
- [6] MySQL bug #83047. Memory usage gradually increases and brings server to halt, 2016. <https://bugs.mysql.com/bug.php?id=83047>.
- [7] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 480–491, New York, NY, USA, 2007. ACM.
- [8] Trishul M. Chilimbi and Matthias Hauswirth. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 156–164, New York, NY, USA, 2004. ACM.
- [9] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. Smoke: Scalable path-sensitive memory leak detection for millions of lines of code. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 72–82, Piscataway, NJ, USA, 2019. IEEE Press.
- [10] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe memory-leak fixing for c programs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 459–470, Piscataway, NJ, USA, 2015. IEEE Press.
- [11] GCC. Labels as values. <https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html>.
- [12] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. Eio: Error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST '08, USA, 2008. USENIX Association.
- [13] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 310–323, New York, NY, USA, 2005. ACM.
- [14] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.
- [15] David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 168–181, New York, NY, USA, 2003. ACM.
- [16] David L. Heine and Monica S. Lam. Static detection of leaks in polymorphic containers. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 252–261, New York, NY, USA, 2006. ACM.
- [17] Jian Huang, Michael Allen-Bond, and Xuechen Zhang. Pallas: Semantic-aware checking for finding deep bugs in fast path. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 709–722, New York, NY, USA, 2017. ACM.
- [18] Chromium issue 816002. High memory usage in youtube, 2018. <https://bugs.chromium.org/p/chromium/issues/detail?id=816002>.
- [19] Suman Jana, Yuan Kang, Samuel Roth, and Baishakhi Ray. Automatically detecting error handling bugs using error specifications. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 345–362, USA, 2016. USENIX Association.
- [20] Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. Meminsight: Platform-independent memory debugging for javascript. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 345–356, New York, NY, USA, 2015. ACM.

- [21] Zhouyang Jia, Shanshan Li, Tingting Yu, Xiangke Liao, Ji Wang, Xiaodong Liu, and Yunhuai Liu. Detecting error-handling bugs without error specification input. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 213–225. IEEE Press, 2019.
- [22] Maria Jump and Kathryn S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 31–38, New York, NY, USA, 2007. ACM.
- [23] Changhee Jung, Sangho Lee, Easwaran Raman, and Santosh Pande. Automated memory leak detection for production use. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 825–836, New York, NY, USA, 2014. ACM.
- [24] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. See <http://llvm.cs.uiuc.edu>.
- [25] Sangho Lee, Changhee Jung, and Santosh Pande. Detecting memory leaks through introspective dynamic behavior modelling using machine learning. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 814–824, New York, NY, USA, 2014. ACM.
- [26] Mengchen Li, Yuanjun Chen, Linzhang Wang, and Guoqing Xu. Dynamically validating static memory leak warnings. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, page 112–122, New York, NY, USA, 2013. Association for Computing Machinery.
- [27] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Doubletake: Fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 911–922, New York, NY, USA, 2016. Association for Computing Machinery.
- [28] Evan K. Maxwell, Godmar Back, and Naren Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 115–124, New York, NY, USA, 2010. ACM.
- [29] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 577–587, New York, NY, USA, 2014. ACM.
- [30] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 397–407, New York, NY, USA, 2009. ACM.
- [31] Maksim Orlovich and Radu Rugina. Memory leak analysis by contradiction. In *Proceedings of the 13th International Conference on Static Analysis*, SAS'06, pages 405–424. Springer-Verlag, 2006.
- [32] Derek Rayside and Lucy Mendel. Object ownership profiling: A technique for finding and fixing memory leaks. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 194–203, New York, NY, USA, 2007. ACM.
- [33] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.
- [34] Yan Tang, Qi Gao, and Feng Qin. Leaksurvivor: Towards safely tolerating memory leaks for garbage-collected languages. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 307–320, Berkeley, CA, USA, 2008. USENIX Association.
- [35] Yuchi Tian and Baishakhi Ray. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 752–762, New York, NY, USA, 2017. Association for Computing Machinery.
- [36] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA, August 2014. USENIX Association.
- [37] Nginx ticket #1482. Memory leak in error handling block in ngx_stream_geo_block method, 2018. <https://trac.nginx.org/nginx/ticket/1482>.
- [38] John Vilks and Emery D. Berger. Bleak: Automatically debugging memory leaks in web applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 15–29, New York, NY, USA, 2018. ACM.
- [39] Common Vulnerabilities and Exposures. Cve-2019-12379, 2019. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12379>.

- [40] Common Vulnerabilities and Exposures. Cve-2019-8980, 2019. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-8980>.
- [41] Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. Enhancing cross-isa dbt through automatically learned translation rules. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '18*, pages 84–97, New York, NY, USA, 2018. Association for Computing Machinery.
- [42] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving integer security for systems with KINT. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 163–177, Hollywood, CA, 2012. USENIX.
- [43] Mingyuan Xia, Wenbo He, Xue Liu, and Jie Liu. Why application errors drain battery easily?: A study of memory leaks in smartphone apps. In *Proceedings of the Workshop on Power-Aware Computing and Systems, Hot-Power '13*, pages 2:1–2:5, New York, NY, USA, 2013. ACM.
- [44] Yichen Xie and Alex Aiken. Context- and path-sensitive memory leak detection. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE '05*, pages 115–125, New York, NY, USA, 2005. ACM.
- [45] Guoqing Xu and Atanas Rountev. Precise memory leak detection for java software using container profiling. In *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, pages 151–160, New York, NY, USA, 2008. ACM.
- [46] Dacong Yan, Guoqing Xu, Shengqian Yang, and Atanas Rountev. Leakchecker: Practical static memory leak detection for managed languages. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14*, pages 87:87–87:97, New York, NY, USA, 2014. ACM.

Argus: Debugging Performance Issues in Modern Desktop Applications with Annotated Causal Tracing

Lingmei Weng[†]

Peng Huang[‡]

Jason Nieh[†]

Junfeng Yang[†]

Columbia University[†]

Johns Hopkins University[‡]

Abstract

Modern desktop applications involve many asynchronous, concurrent interactions that make performance issues difficult to diagnose. Although prior work has used causal tracing for debugging performance issues in distributed systems, we find that these techniques suffer from high inaccuracies for desktop applications. We present Argus, a fast, effective causal tracing tool for debugging performance anomalies in desktop applications. Argus introduces a novel notion of strong and weak edges to explicitly model and annotate trace graph ambiguities, a new beam-search-based diagnosis algorithm to select the most likely causal paths in the presence of ambiguities, and a new way to compare causal paths across normal and abnormal executions. We have implemented Argus across multiple versions of macOS and evaluated it on 12 infamous spinning pinwheel issues in popular macOS applications. Argus diagnosed the root causes for all issues, 10 of which were previously unknown, some of which have been open for several years. Argus incurs less than 5% CPU overhead when its system-wide tracing is enabled, making always-on tracing feasible.

1 Introduction

Diagnosing performance anomalies is an essential need for all kinds of software. For modern desktop applications, performance diagnosis can be very difficult. Such applications are often built with assorted frameworks and libraries. For responsiveness, they divide handling of user interface (UI) events into many small execution segments [30] that run concurrently on multi-core hardware. For instance, macOS applications handle UI events by sending messages to delegate objects that contain code to react to these events asynchronously. The messages are generated by the closed-source Cocoa framework [11], which in turn interacts with the operating system (OS), daemons, and other libraries. The asynchronous, predominantly concurrent interactions obscure the true cause of a performance anomaly.

Traditional debugging and profiling tools are not well suited to troubleshoot performance issues in desktop applications. macOS tools such as `spindump` [10] and `lldb` [45] allow users to analyze a buggy process' stack traces. Profilers like `Gprof` [28], `perf` [5], and macOS Instruments [12] mainly analyze what functions take the most time. None of these tools provide insights regarding the sequence of events that

span across the many frameworks, libraries, system daemons, kernel, application processes/threads, and result in the performance issue. Traditional tools excel at analyzing system state at a specific point in time in an individual component. They are not amenable to analyzing concurrent execution flows over time whose interactions may cause performance issues.

To debug cross-component performance issues, causal tracing has been proposed [14, 20, 27, 32, 37, 38, 40, 41, 43, 44, 46, 56], especially for distributed systems. Causal tracing utilizes a trace graph to help developers understand performance issues that involve complex interactions. A trace graph consists of vertices and edges, where vertices are *execution segments*, such as an operation, system event, message, etc., and edges indicate causal relationships between vertices. To diagnose a performance issue, these solutions usually run a critical path analysis on the constructed trace graph that finds the sequence of vertices and edges which start from the vertex where the problem occurs and take the greatest amount of time for completion.

Unfortunately, we observe that previous causal tracing approaches are ineffective for desktop applications because they cannot accurately identify the boundaries of execution segments and their causality relationships. For example, a long-standing Google Chrome web browser performance anomaly [2] on macOS occurs when a user enters non-English words in the search box, causing Chrome to hang with the infamous macOS spinning pinwheel, which appears when an application is not responsive to user input. Using previous approaches to construct trace graphs for the multi-threaded, multi-process browser results in many missing execution segments and many additional irrelevant execution segments. Attempting to diagnose the problem using these incomplete and inaccurate graphs would incorrectly pinpoint no events or wrong events as the culprit. In theory, these tracing inaccuracies could be fixed by adding instrumentation, such as adding constraints in noisy trace points to filter irrelevant events. However, frameworks and libraries used by desktop applications have diverse programming idioms and are often closed-source, making deep instrumentation difficult. Extensive instrumentation would also incur prohibitive overhead, resulting in unacceptable performance.

To address these problems, we have created Argus, a causal tracing tool specially designed to help users diagnose performance anomalies in desktop applications. Argus is

based on the insight that tracing inaccuracies are inherently unavoidable in real desktop systems, so instead of trying to eliminate all inaccuracies, we should design tracing solutions that can accommodate some inaccuracies. Argus introduces a new notion of annotated trace graphs, in which edges are explicitly annotated as *strong* and *weak* edges. Strong edges represent connections among segments based on typical programming paradigms that must be causal, such as sending and receiving an IPC message. Weak edges represent ambiguous relationships among segments. For example, when one thread wakes up another thread, it could be a causal relation, e.g., `lock/unlock`, or just an artifact of regular OS scheduling. Argus further boosts or prunes unnecessary weak edges by leveraging operation semantics and call stacks.

Argus introduces a new beam search diagnosis algorithm based on edge strength and a novel method of comparing trace subgraphs across normal and abnormal executions of an application. The algorithm is motivated by our observation that critical path analysis used in prior work is ineffective due to inaccuracies inherent in trace graphs. Beam search embraces more possibilities while exploring the annotated noisy trace graph. Our algorithm efficiently selects likely causal paths in the massive trace graph and tolerates noises. Comparing trace subgraphs across normal and abnormal executions also helps with diagnosis when the problem is due to missing operations in the abnormal execution.

Argus provides system-wide tracing by extending existing tracing support in the OS kernel and applying binary patching for low-level libraries. This allows Argus to easily track objects across process boundaries, account for kernel threads involved in communications among processes, and cover customized programming paradigms by operating in a common low-level substrate used by higher-level synchronization methods and APIs that may be introduced and evolve over time. Argus does not require any application modifications.

We have implemented and evaluated a prototype of Argus across multiple versions of macOS. This presents a harsh test for Argus given the many complex, closed-source frameworks, libraries, and applications in the macOS software stack. We evaluated Argus on 12 real-world spinning pinwheel issues in widely-used macOS applications, such as Chrome, Inkscape, and VLC. Argus successfully pinpoints the root cause and sequence of culprit events for all cases. This result is particularly notable given that 10 of the 12 cases are open issues whose root causes were previously unknown to developers. Argus incurs runtime overhead low enough such that users can leave Argus tracing always-on in production without experiencing any noticeable performance degradation. Source code for Argus is available at <https://github.com/columbia/ArgusDebugger>.

2 Motivation and Observations

We experienced first-hand the Chrome web browser performance issue on macOS. Typing non-English words in a search

box while a web page is loading causes Chrome to freeze and trigger a spinning pinwheel. The spinning pinwheel appears when an application is not responsive to user input for more than two seconds. Others have also experienced this issue with the Chromium web browser and reported it to Chromium developers [2]; Chrome is based on Chromium.

We study the bug in Chromium since it is open-source, so we can verify its ground truth. Chromium is a multi-process macOS application involving a browser process and several renderer processes, each process having dozens of threads. When a user types a string in the browser search box, a thread in the browser process sends an IPC message to a thread in the renderer process, where the rendering view code runs to calculate the bounding box of the string, which in turn queries `fontd`, the font service daemon, for font dimensions.

To diagnose the bug, we first tried using `spindump` [10], a widely-used macOS debugging tool, which shows the main thread of the browser process is blocking on a condition variable. However, `spindump` provides no clue as to why the condition variable is not signaled. Using macOS Instruments [12] was also ineffective, as it simply analyzes what functions take the most time, which are not the root cause in this case. These traditional debugging and profiling tools are fundamentally not well suited to analyzing causality in highly concurrent execution flows across multiple components over time.

We next tried state-of-the-art causal tracing techniques. Specifically, we use Panappticon [56], a system-wide tracing tool originally built for Android. We reimplemented a version for macOS with more complete tracing of asynchronous tasks, using non-intrusive interposition to trace asynchronous tasks, IPCs, and thread synchronizations from the system and libraries. We use the tool when running Chromium and reproduce the anomaly by typing non-English search strings. After the browser handles the first few characters normally, the remaining characters trigger a spinning pinwheel. We then stop the tracing. The entire session took around five minutes.

Dividing up the trace graph into separate graphs each beginning from a user input event results in 359 trace graphs; user input events are dispatched from the macOS `WindowServer` process to Chromium. The trace graphs are highly complex, with 888,236 vertices and 751,332 edges in total. They span across 11 applications, 79 daemons including `fontd`, `mdworker`, `nsurlsessiond`, and various helper tools started by the applications. They cover 90 processes, 1177 threads, and 644K IPC messages.

Studying the trace graphs, we observe: (i) connections exist between graphs from different UI events; (ii) some long execution segments have no boundaries; (iii) there are orphaned vertices with no edges; (iv) the trace graph that contains the anomalous event sequence triggering the spinning pinwheel contains 12 processes—3 are clearly unrelated to the transaction, and 6 are daemons whose relationships are unclear without further investigation. Based on further analysis of these graphs with call stacks and reverse

```

1 // worker thread in fontd:
2 block = dispatch_mig_server;
3 dispatch_async(block);

1 // implementation of dispatch_mig_server
2 dispatch_mig_server()
3 for (;;) { // batch processing
4     mach_msg(send_reply,recv_request)
5     call_back(recv_request)
6     set_reply(send_reply)
7 }

```

Figure 1: Dispatch message batching. `dispatch_mig_server` can serve unrelated applications together.

engineering techniques, we conclude that they have significant inaccuracies. Running diagnosis on them leads to a wild goose chase, investigating components such as `fontd`, as it sends out messages after a long execution, which turn out to be completely unrelated to the root cause. We observe two general inaccuracies: *over-connections* and *under-connections*.

Over-connections usually occur when intra-thread execution segment boundaries are missing. We summarize three common programming patterns responsible for this—dispatch message batching, piggyback optimization, and superfluous wake-ups.

Dispatch message batching. Frameworks and daemons often implement event loops for handling multiple events inside callback functions. For example, Figure 1 shows two threads from the `fontd` daemon in macOS; the worker thread installs a callback function `dispatch_mig_server()` in a dispatch queue and the main thread dequeues and calls the function via `dispatch_client_callout`. `dispatch_mig_server()` has an event loop which batch processes requests from different applications, presumably for performance. It invokes `call_back` to process a message and `set_reply` to post a reply. However, previous causal tracing tools like Panappticon assume the execution of a callback function is entirely on behalf of one request. `dispatch_mig_server` is thus treated as a single execution segment and edges are added between the vertex representing `dispatch_mig_server` and the many unrelated applications for which it handles requests. These edges incorrectly indicate causal relationships that would result in misleading diagnoses.

Piggyback optimization. Frameworks and daemons may piggyback multiple tasks in a system call to reduce kernel boundary crossings. For example, Figure 2 shows the macOS system daemon `WindowServer` uses a single system call `mach_msg_overwrite` to receive data and piggyback the reply for an unrelated event. However, previous causal tracing tools like Panappticon treat the execution of a system call as a single execution segment for one event, artificially making many events appear causally related.

Non-causal wake-up. Desktop applications typically have multiple threads synchronized via mutual exclusion, such that a thread’s unlock operation wakes up another waiting thread. Such a wake-up may be, but is not always, intended as causality. For example, in Chromium, a wake-up is commonly followed by a batch processing block, but it is unclear whether

```

1 //a thread in WindowServer
2 while (true){
3     //postpone a reply
4     CGXPostReplyMessage(msg);
5     //receive requests
6     CGXRunOneServicePass();
7 }

8 CGXRunOneServicePass(){
9     if (_gOutMsgPending)
10        mach_msg_overwrite(
11            SEND|RECV,
12            _gOutMsg, RecvMsg)
13        else
14            mach_msg(RECV,RecvMsg)
15 }

```

Figure 2: Piggyback optimization and intra-thread data dependency. `mach_msg_overwrite` combines the reply of a previous event. Operations inside a thread have dependencies on `_gOutMsg`.

```

1 // worker thread needs
2 // UI update
3 obj->need_display = 1

1 //main thread
2 if (obj->need_display == 1)
3     render(obj)

```

Figure 3: Shared data flag across threads.

the following events being batch processed depend on the wake-up event. Previous causal tracing tools assume any wake-up is causal, which may artificially make events appear causally related when they are not.

Under-connections usually occur due to missing intra-thread data dependencies and inter-thread shared flags.

Data dependency. Frameworks and daemons may have internal state that causally link different execution segments of a thread. For example, Figure 2 shows that a `WindowServer` thread calls the function `CGXPostReplyMessage` to save the reply message, which it internally stores in a variable `_gOutMsg`. When the thread later calls `CGXRunOneServicePass`, it sends out `_gOutMsg` if there is any pending message.

Shared data flags. Frameworks and daemons may use shared flags that causally link different threads. Figure 3 shows a worker thread sets a field `need_display` inside a `CoreAnimation` object whenever the object needs to be repainted. The main thread iterates over all animation objects and reads this flag, rendering any such object. Existing tools do not track these kinds of shared-memory communication.

3 Overview of Argus

We have designed Argus to diagnose performance issues in desktop applications. Argus satisfies four key requirements not met by previous causal tracing tools: (1) use minimal instrumentation, (2) support closed-source components, (3) extract rich information from heterogeneous components with minimal manual effort, and (4) incur low runtime overhead.

Central to its design is the construction of *annotated trace graphs* from low-level trace events. Argus introduces the notion of *strong* and *weak* edges in trace graphs to mitigate inherent inaccuracies in tracing. When there is strong evidence of causality, such as an IPC message event, Argus adds a *strong edge* between vertices. When an execution segment is created by events that may not necessarily represent causality, such as non-causal wake-ups, Argus adds a *weak edge*. During diagnosis, Argus prefers traversing through strong edges when possible. Argus also stores extra semantic information in the graph vertices, including user input events, system calls, and

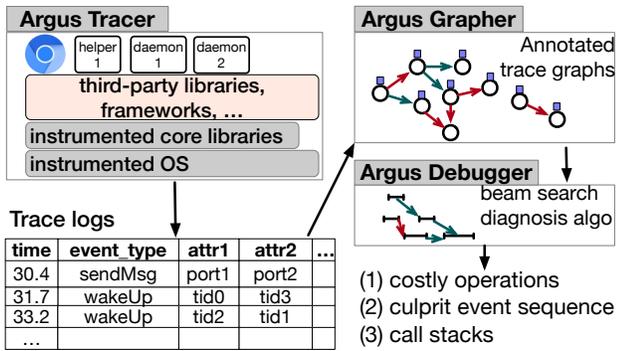


Figure 4: Overview of Argus.

sampled call stacks. This extra information is used to improve weak edge annotation and align and compare trace graphs for normal and abnormal execution to aid diagnosis.

Figure 4 shows an overview of Argus. It consists of three main components—a tracer, a grapher, and a debugger. The tracer runs continuously in the background on a user’s machine, transparently logging events from low-level system libraries and the kernel, without any need to modify applications. When a user encounters some performance anomaly, she reports the issue about the problematic application, along with the timestamp of the anomaly occurrence. The reported issue and trace logs are sent to the developer, the logs containing events for both normal execution and abnormal execution when the performance anomaly occurs. The developer feeds the logs into the grapher to construct the annotated trace graphs for both normal and abnormal execution, and runs the debugger on the graphs to output the diagnosis results.

4 Argus Tracer

Argus traces events inside the kernel and low-level libraries, with minimal instrumentation. This provides three advantages over tracing in user applications. First, tracing in the kernel and libraries ensures coverage of custom programming paradigms. For instance, Argus traces general thread scheduling events and wake-up and wait to ensure coverage of a variety of custom synchronization primitives in desktop applications, because their implementations almost always use kernel wake-up and wait. Second, tracing in the kernel helps connect tracing events across process boundaries, because the addresses of the traced objects in kernel space are usually unique, while tracing in user programs requires maintaining and propagating unique identifiers. Third, tracing kernel threads helps bridge communications among processes. For instance, a kernel thread sends out a message to a process when the process needs to execute a delayed function.

In the macOS XNU kernel, Argus traces system calls, thread scheduling, interrupts, time-delayed calls, and Mach messages. Argus leverages existing macOS kernel tracing support [13], but adds enhancements to log more information and enable always-on tracing using a ring buffer to avoid exhausting

storage. The enhancements require roughly 500 lines of code (LOC) in the XNU kernel, which are straightforward to add given that the kernel is open source. Trace events are asynchronously flushed to a file with a size limit. The limit is by default 2 GB, which can store roughly 20 million trace events; this is about 5 minutes of tracing when running large applications like Chrome. It can be easily adjusted to accommodate longer execution times. We used the default limit for all experiments in Section 7.

Argus logs kernel events to identify when threads are executing and their causal relationships. All system calls are traced to provide high-level semantics that can be used to identify causal relationships. Argus simply records return values for most system calls, but call stacks are also logged for a small set of system calls, namely those pertaining to Mach messages and synchronization using conditional variables and semaphores. Call stack information is later used by the Argus debugger to provide debugging information for developers. Thread scheduling is traced to track when a thread becomes idle and which thread wakes it up. Argus logs three types of thread scheduling events: *wait* to indicate when a thread becomes idle, *wake-up* to indicate when the current thread wakes up another thread, and *preempt* to indicate when a thread is preempted due to its timeslice being used up or priority policies. Interrupts are logged to indicate when threads are preempted by interrupts, with call stacks also logged for interprocessor interrupts (IPIs). Argus traces the internal kernel implementation of time-delayed calls, which are used to implement asynchronous calls in libraries such as Grand Central Dispatch (GCD). Finally, Argus traces the internal kernel implementation of Mach messages, not just their invocation via system calls, to enabling tracing of all use of Mach messages, including use within the kernel among kernel threads.

To aid developers in interpreting the virtual addresses in call stacks via `lldb`, Argus also logs in userspace the virtual memory layout of images for all processes. The tracer records the virtual memory maps for all running processes when tracing is enabled or terminated; processes launched during tracing are also recorded. The memory layout information is also fed to the Argus debugger.

In addition to kernel tracing, Argus traces four closed-source macOS frameworks, `AppKit`, `libdispatch.dylib`, `CoreFoundation`, and `CoreGraphics`, to track UI events and batch processing paradigms used by applications. Because these frameworks are closed source, the trace events are added via binary instrumentation using a mechanism similar to `Detour` [31]. `AppKit` is used to dispatch UI events to handlers. Argus traces where a UI event is fetched from the `WindowServer` and dispatched to an event handler. `libdispatch.dylib` implements GCD, managing dispatch queues to balance work across the entire system. Argus adds trace events to track when objects are pushed into a dispatch queue and popped off of the dispatch queue and executed. `CoreFoundation` supports event loops for GUI applications, which are widely used to process requests

from timers, customized observers, and sources such as sockets, ports, and files. Argus adds trace events so the handling of different requests inside event loops can be tracked separately.

To deal with the under-connection issues (Section 2), we annotate a handful of data flags in CoreGraphics. Given the shared flag variable names, Argus monitors the respective virtual addresses with watchpoint registers. Reads or writes to the addresses will invoke a signal handler that records trace events with the values stored in those addresses. Argus adds code to CoreFoundation to install this signal handler.

Argus can use the same watchpoint mechanism to trace shared data flags in applications. To assist developers in finding these shared data flags, Argus provides a lightweight tool that uses lldb to record the operand values of each instruction and finds ones that lead to divergence in control flow, which are likely data flags. The shared flag variable names are recorded in an Argus tracer configuration file, which are then traced using the same signal handler installed by CoreFoundation. Since CoreFoundation is imported by all GUI applications, Argus can trace these shared data flag accesses without any application modifications.

Note that the annotation effort for shared data flags is in general small. This is because execution segments that access shared variables are usually connected already by some types of causality, e.g., wait/signal events; developers mainly need to provide Argus with shared flags that are accessed through ad-hoc synchronization [49]. In our experience, only a few shared flags need to be monitored. Also for this reason, although hardware watchpoint registers are limited, Argus is unlikely to exhaust them. In fact, none of the applications we evaluated in Section 7 needed shared flags to be identified or traced in the applications themselves. Mechanisms such as Kprobe [3] could potentially be used to extend Argus to support monitoring more shared flags.

5 Argus Grapher

Argus uses the trace logs to build an annotated trace graph by first identifying the boundaries of execution segments in each thread to determine the graph vertices, then adding annotated edges between vertices. The annotated edges contain type metadata to indicate *strong* versus *weak* edges, which is used during diagnosis to mitigate inaccuracies due to over-connections and under-connections, as discussed in Section 2.

Argus first determines the execution segments that will form the graph vertices. Using various trace events as boundaries, Argus splits the execution of each thread into separate execution segments. First, Argus splits nesting of tasks executed from dispatch queues. If an execution of `dispatch_callout` invokes several other `dispatch_callout`, each dispatched task is separated. Second, Argus recognizes batch processing patterns such as `dispatch_mig_server()` in Figure 1 and splits the batch into separate execution segments. Third, when a wait operation blocks a thread execution, Argus splits the execution

Edge	Rules for Edge Annotation
Strong	1. IPC message send and receive; 2. Asynchronous calls (work queue, delayed call); 3. Direct wake-up of a thread on purpose; 4. Data dependency.
Weak	1. Non-causal wake-up; 2. Execution segments divided between a wait event and a wake-up event, excluding following cases: wait or wakeup are introduced by system call <code>workq_kern_return</code> , or they are in <code>kern_task</code> ; 3. Split suspicious batching execution segments, except known batching APIs: <code>RunLoopDoObservers</code> , <code>CGXServer</code> , etc.
Boosted Weak	Continuous execution segments matching weak edge rules but are on behalf of the same task.

Table 1: Edge annotation rules.

into separate segments at the entry of the blocking wait. The rationale is that blocking wait is typically done as the last step in event processing. Finally, Argus uses Mach messages to split execution when the set of communicating peers differs. Argus maintains a set of peers, including the direct sender or receiver of the message and the beneficiary of the message; macOS allows a process to send or receive messages on behalf of a third process. Argus splits execution when two consecutive messages have non-overlapping peer sets. By splitting thread execution using these four criteria, Argus avoids potential over-connections due to batching and piggyback optimizations.

Argus next determines the edges that should be added between vertices. Edges are introduced to reveal the causality of two execution segments and thus guide the causal path exploration. Based on the rules in Table 1, Argus annotates three types of edges: *strong*, *weak*, and *boosted weak*.

First, Argus adds strong edges by identifying Mach message, dispatch queue, time-delayed call, and data flag trace events associated with a vertex and finding the corresponding peer events and peer vertices. For Mach message events, Argus adds a strong edge from the vertex with the message send event to the vertex with its associated receive event. If a message requires a reply, the received message can produce a reply message, which can be sent by a third thread, in which case Argus adds a strong edge from the vertex with the received message event to the one with the send event for the reply message. For dispatch queue events, Argus adds a strong edge from the vertex where the callback function is pushed to a dispatch queue to the vertex where the callback function is invoked. For time-delayed calls, Argus adds a strong edge from the vertex where the timer is armed to the vertex where the callback function is fired. For shared data flags, Argus adds a strong edge from the vertex with a data flag write event to the vertex with its corresponding read event, avoiding potential under-connections.

Second, Argus adds edges by identifying thread scheduling trace events and finding the events and vertices corresponding to the pair operations. Argus adds strong edges only when the context clearly indicates causality, such as the signal and wait operations of a condition variable. Otherwise, Argus adds only weak edges. One hint Argus takes from macOS is

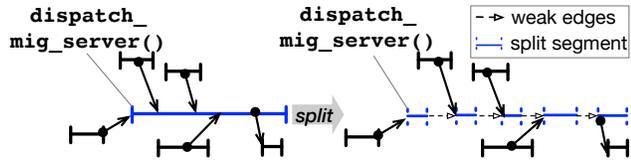


Figure 5: The segment for batch processing in `dispatch_mig_server` is split into multiple segments to distinguish different items. Weak edges are added among the split segments.

that, if a wake-up is not followed by a specific communication operation (e.g., message receive), and does not target a specific thread but all threads on the wait queue, then it is likely not causal, in which case a weak edge is added.

Third, because Argus splits the execution of a thread into segments (graph vertices) based on heuristics that may not always be valid, Argus adds weak edges between these adjacent execution segments, as shown in Figure 5. Argus converts a weak edge into a *boosted weak edge* if two continuous execution segments are on behalf of the same task. It infers whether the segments are for the same task by leveraging call stack symbols. We calculate frequencies for all symbols across the whole tracing and notice a low-frequency (bottom 10%) symbol usually only appears in a task from a specific application, compared to high-frequency symbols from system routines or framework APIs. Thus, if the two segments share the same low-frequency symbols, Argus infers they are collaborating on the same task and sets a boost flag for the weak edge between them.

However, abuse of weak edges could generate excessive false positives during diagnosis, so Argus takes advantage of high-level semantics to avoid adding unnecessary weak edges between adjacent execution segments. First, if the call stacks of two segments of a thread share no common symbols or share a recognized system library batching API, Argus does not add a weak edge between them. Second, because wait and wake-up events are mostly from system calls, Argus leverages system call semantics to determine the necessity of weak edges. For example, we find the wait event from system call `workq_kern_return` indicates an end of a task in the thread, while the wake-up event formed in `workq_kern_return` intends to acquire more worker threads for concurrent tasks in the dispatch queue. Execution segments containing such event sequences do not need bridging with weak edges. Finally, the kernel task in macOS acts as a delegate to provide service for many applications, such as I/O processing and timed delayed invocations. The kernel task threads contain execution segments beginning with a wake-up event and ending with a wait event. Each segment serves different requests and they are not causally related, so weak edges are not added between those kernel task execution segments.

6 Argus Debugger

Argus uses the constructed trace graphs to diagnose performance issues by starting with the vertex that contains the

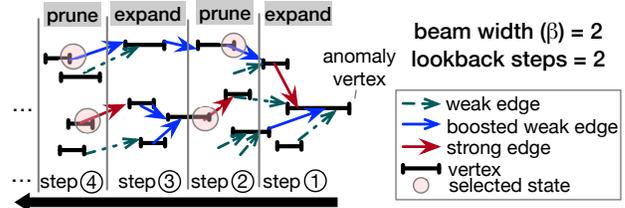


Figure 6: Beam search diagnosis algorithm. Search backwards from the anomaly vertex; choose the best β states to expand next. For every lookback steps, prune the existing states to at most β paths.

performance anomaly and traversing the graphs to identify the causal paths including the root cause vertices. The typical critical path analysis used in existing causal tracing solutions cannot effectively handle the noises in the trace graphs. Argus introduces a new diagnosis algorithm based on *beam search* to efficiently explore the causal paths likely related to the performance anomalies. It also introduces a novel subgraph comparison mechanism to find missing vertices not present in the trace graph for abnormal execution that are present in the graph for normal execution. This comparison is helpful to identify the root cause that would be otherwise unknown.

6.1 Causal Path Search—Beam Search

From a given vertex that contains the anomaly, such as the spinning cursor, Argus finds what path “caused” the anomaly by using beam search based on a cost function for annotated edges. Beam search is similar to breadth-first-search, but at each search step, it sorts the next level of graph vertices based on a cost function and only stores β —the beam width—best vertices to consider next. Argus customizes its beam search with a *lookback* scheme such that the algorithm evaluates the cost function for multiple levels of edges before pruning. Argus evaluates the vertices and prunes them with β only after the search advances the configured *lookback* steps to avoiding pruning paths with weak edges too early.

Argus’s beam search algorithm provides two key advantages. First, compared to brute-force search, beam search only explores the most promising vertices, which is essential given that trace graphs are highly complex with millions of edges; searching all paths would be too inefficient and, given graph inaccuracies, result in an overwhelming number of options to consider. Second compared to local search methods such as hill-climbing, beam search embraces more possible causal paths because it ranks partial solutions and the ranking changes during the exploration. For example, assuming strong edges are preferred to weak ones, a path with a weak edge followed by a series of strong edges is likely to get a higher ranking and be returned by beam search, but will be missed by a hill-climbing search algorithm.

Figure 6 illustrates the algorithm. It searches for causal paths *backwards* from the anomaly vertex. For each incoming edge of the current vertex, the algorithm computes the *penalty score* for the new path. At every lookback step, the search branches

Algorithm 1: Causal Path Search Algorithm (Beam Search).

Data: g - event graphs, $curVertex$ - vertex inspected in current search state, $beamWidth$ - search branches at most, $lookbackSteps$ - searching steps taken before pruning current search branches

Result: paths

```

1 Function BeamSearch( $g, curVertex, beamWidth, lookbackSteps$ ):
2    $curStates.init(curVertex)$ ;
3    $curSteps \leftarrow 0$ ;
4   while  $curStates.incoming\_edges() > 0 \ \&\& \ beamWidth > 0$  do
5      $++curSteps$ ;
6      $newStates.clear()$ ;
7     for each  $state \in curStates$  do
8       if  $beamWidth \leq 0$  then
9          $break$ ;
10      end
11      if  $state.path.reach(UI) \ || \ state.path.incoming\_edges = 0$ 
12        then
13           $paths.add(state.path)$ ;
14           $--beamWidth$ ;
15        end
16        for each  $edge \in state.path.incoming\_edges$  do
17           $newState.path \leftarrow state.path + edge$ ;
18           $newState.score \leftarrow state.score + penalty(edge.val)$ ;
19           $newStates.add(newState)$ ;
20        end
21         $curStates \leftarrow newStates$ ;
22        if  $curSteps = lookbackSteps$  then
23           $pruneStates(curStates, beamWidth)$ ;
24           $curSteps \leftarrow 0$ ;
25        end
26      end
27       $pruneStates(curStates, beamWidth)$ ;
28       $paths.append(curStates.paths)$ ;
29      return  $SortIncPenaltyScore(paths)$ ;
30 Function  $pruneStates(newStates, beamWidth)$ :
31    $SortIncPenaltyScore(newStates.paths)$ ;
32   while  $newStates.size() > beamWidth$  do
33      $newStates.pop\_back()$ ;
34   end
35   return;

```

are pruned: it sorts the paths by their penalty scores and only retains at most β paths with low penalties. A path is added to the result if a vertex is reached containing a UI event or has no incoming edges, and the beam width decreases by one. Using such vertices as for path termination helps developers understand causality in an end-to-end request handling transaction.

Algorithm 1 lists the pseudo-code of the search algorithm. Lines 16 – 18 compute penalty scores for new paths after incoming edges are added to the path. Lines 22 – 25 prune the searched branches every L lookback steps. Paths are sorted by their penalty scores and paths with high penalties are discarded. Penalty scores are calculated with a linear function on edge values, where a strong edge is -1, a weak edge is 1, and a boosted weak edge is 0. A path with n edges has a penalty $p = \sum_{i=1}^n (a \times E_i + b)$, where E_i is the i th edge value. This approach guides search towards paths with stronger causality. While more complex non-linear functions may be feasible, this simple function works well for many diagnosis cases.

The beam width setting affects the search efficiency and diagnosis accuracy. A setting too large would cause path explosion and noisy paths to be returned. A setting too small may easily miss the true causal path. We set $\beta = 5$ to strike a good balance. Tuning this parameter is relatively easy in practice. The lookback step setting is set based on observing

Algorithm 2: Subgraph Comparison Algorithm.

Data: $anomVertex$ – problematic vertex, $anomGraph$ – trace graph for anomaly case, $normGraph$ – trace graph for normal case

Result: ret- potential culprits of anomaly

```

1 Function SubGraphCompare( $anomVertex, anomGraph, normGraph$ ):
2    $ret.clear()$ ;
3    $similarVertices \leftarrow FindSimilarVertices(normGraph, anomVertex)$ ;
4    $baselineVertex \leftarrow GetBaseLine(similarVertices, anomVertex)$ ;
5    $targetVertex \leftarrow woken(normGraph, baselineVertex)$ ;
6    $causalPaths \leftarrow BeamSearch(normGraph, targetVertex, beamWidth, lookbackStep)$ ;
7   // sub-graph is constituted with paths;
8   for each  $causalPath \in causalPaths$  do
9     for each  $vertex \in causalPath$  do
10       $expectVertex \leftarrow SimilarVertex(anomGraph, vertex)$ ;
11      if  $expectVertex = \emptyset$  then
12        // missing similarity to vertex;
13         $anomThr \leftarrow SearchThread(anomGraph, vertex.thread)$ ;
14        // get the vertex that causes the dissimilar;
15         $suspVertex \leftarrow VertexInThread(anomGraph, anomThr)$ ;
16      else if  $DifferentVertices(expectVertex, vertex)$  then
17        // vertex acts different from normal case;
18         $suspVertex \leftarrow expectVertex$ ;
19      else
20         $continue$ ;
21      end
22       $ret.push\_back(suspVertex)$ ;
23    end
24    if  $!ret.empty()$  then
25       $return ret$ ;
26    end
27  end
28  return  $ret$ ;

```

that traversal of most graphs encounters a weak edge within five steps. We set $L = 5$ to tolerate weak edges. Given this setting, a path of x strong edges, y weak edges, and z boosted weak edges has a penalty of $p = -a \times (x - y) + 5 \times b$. If all edges are strong, the penalty is negative only when $b < a$. If there are weak edges, the penalty is positive only when $(x - y) \times a < 5 \times b$, where $-3 < x - y < 3$. Therefore, we set the default penalty function coefficients $a = 3$ and $b = 2$.

6.2 Subgraph Comparison

If we run causality analysis only on the trace graph constructed with the anomalous performance issue, the root cause may not be exposed in some cases. For example, a blocked function could be caused by a missing wake-up from one of the background threads. If the thread does not perform the wake-up during abnormal execution, there will be no execution segment with the wake-up, and therefore no vertex in the anomalous trace graph that can be identified correctly as the root cause. Argus addresses this problem by first constructing the trace graphs for both normal and abnormal execution. It then uses its beam search method on the normal trace graph to identify the causal paths in that graph that corresponds to the desired normal behavior that does not occur during abnormal execution. We refer to those causal paths a *subgraph*. Argus then uses the vertices in the subgraph to identify the missing root cause in the abnormal execution. This is done by introducing a novel sub-

graph comparison method between the trace graphs for both normal and abnormal execution, which is listed in Algorithm 2.

Argus first determines a baseline vertex in the normal graph that is comparable to the anomaly vertex in the anomalous graph. Argus computes a signature for each vertex based on the trace event sequence in its execution segment. The signature is composed of two parts, one that encodes the types corresponding to the event sequence e.g. 0 for IPC event, 1 for syscall event, etc., and another that is a hash of the event parameters, e.g., process names of IPC events. Argus calculates the similarity of two vertex signatures using string edit distance. Among the vertices in the normal graph that are similar to the anomaly vertex, Argus chooses one that behaves differently from the anomaly vertex, based on return values of system calls and execution times. For example, a vertex whose last event is a blocking system call with a timed wait may behave in two different ways, timing out or quickly woken up.

After Argus identifies a baseline vertex, it obtains its causal paths using Algorithm 1. The result is a subgraph of the normal trace graph rooted from the baseline vertex to some ending vertex. Argus examines the subgraph from the most related causal path. Starting with the ending vertex V , whose execution segment was executed by some thread T , Argus identifies vertices in the abnormal trace graph that were also executed by T . For each identified vertex, Argus checks whether it behaves differently from V , in which case it is flagged as a suspicious vertex. If no such vertices are found, Argus repeats this procedure with the next vertex in the subgraph. Otherwise, for each suspicious vertex that has incoming edges, Argus recursively repeats the subgraph comparison by treating the suspicious vertex as the initial anomaly vertex. The recursive procedure effectively keeps working backwards through vertices to eventually find a set of root cause candidate vertices in the anomalous trace graph with no incoming edges. Argus then returns the vertex whose path to the original anomalous vertex has the lowest penalty score, identifying that vertex as the root cause.

Figure 7 shows a simplified example of the subgraph comparison method applied to the Chromium performance issue discussed in Section 2. Vertex E' in the anomalous graph is the initial anomaly vertex. Argus identifies vertex E in the normal graph as having a similar signature but behaving differently, and treats it as a baseline vertex. Argus applies beam search to the normal graph starting with vertex E , resulting in the subgraph $A \leftarrow B \leftarrow C \leftarrow D \leftarrow E$. Argus starts with A , identifies its browser thread, and determines that A cannot be the root cause since the same browser thread contains the performance anomaly E' in the anomalous trace graph. Argus then considers B , identifies its renderer thread, and finds all vertices in the anomalous trace graph executed by the renderer thread. F' is similar to F , so it is not considered a suspicious vertex, but J' is not similar to any vertex in the normal trace graph, so it is considered suspicious. J' has no incoming edges and is identified as a root cause candidate. If there are no other candidates identified, J' is returned as the root cause.

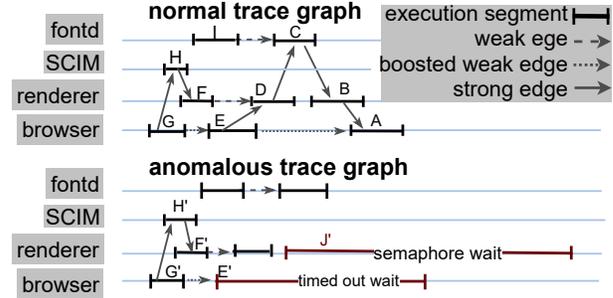


Figure 7: Chromium normal and anomalous trace graphs after user typed in a search box (vertex G/G'). Vertex E' (requesting a bounding box for input) is the anomaly vertex. Sub-graph in normal trace graph is extracted from baseline vertex E . Vertex J' (javascript processing blocks on semaphore) is the root cause Argus reported. Trace graphs are simplified for clarity; only processes are shown and communications with processes such as `imklaunchagent` are omitted.

6.3 Debug Information

Argus further provides the calling contexts of the anomaly vertex and the root cause vertex to help developers localize the bug in code. To do so, Argus examines the call stacks it attaches in the graph vertices. If the anomaly or root cause vertex has a blocking call, the call stack Argus tracer collects would reveal the context of the blocking call directly. If the vertex has a long runtime cost, the problematic vertex usually contains periodic IPIs, where the Argus tracer collects call stacks. In this case, the Argus debugger calculates the longest common sequence of frames from those call stacks. The top frame in the sequence reflects the costly function call.

For instance, in Figure 7, Argus reports the following information: (i) the calling context of problematic vertex E' and its causal path $E' \leftarrow G'$; (ii) the calling context of root cause vertex J' along with its *unmatched causal path* in baseline trace graph: $A \leftarrow B \leftarrow C \leftarrow D \leftarrow E \leftarrow G$, and vertex B is marked because its thread should have woken up the blocking thread in the anomaly case.

6.4 Diagnosis for Spinning Pinwheel in macOS

Argus's debugger can be used to effectively diagnose spinning pinwheel performance issues in macOS applications. Recall that a spinning pinwheel appears when the UI thread of an application can not process any user inputs for over two seconds. During normal execution, the two-second interval may cover many vertices, but when the spinning pinwheel appears, the main thread of the application is stalled and the two-second interval covers only a single vertex. Leveraging this timing information, Argus identifies the anomaly vertex in the main thread of the targeted application and classifies the issue as either a *LongRunning* and *LongWait* anomaly.

LongRunning. The main thread is busy performing lengthy CPU operations and therefore its execution segment is in the anomalous trace graph. Argus uses its beam search method

to identify the causal path between the anomaly vertex and the vertex with the UI event resulting in the issue. Argus reports the costly API, event handler, and causal path to the developer.

LongWait. A UI thread is blocked, but it is hard to tell why. Argus uses its subgraph comparison method together with its beam search method to deduce which vertex is missing from the anomalous trace graph. A long-wait event could be caused by another long-wait event. Argus supports recursively diagnosing “the culprit of the culprit.” Therefore, it can reveal deep root causes. At the end of each iteration of diagnosis, the calling context of problematic vertex, root cause vertices in the anomalous trace graph, and causal paths are ranked and reported to users.

Some LongRunning issues may be diagnosed with existing tools such as `spindump` if the profiling is accurate and complete. However, Argus is better in that a call stack is usually not enough to connect the busy processing to the event handler, due to the prevalence of asynchronous calls. Also, call stack profiles after the anomaly may miss the real costly operations. LongWait issues usually involve multiple components and are extremely hard to understand and fix with current tools. Those issues may remain unresolved for years and significantly hurt user experience and developer productivity.

7 Evaluation

We have implemented Argus across multiple versions of macOS, ranging from El Capitan to Catalina. We evaluate Argus to answer several key research questions: (1) Can Argus effectively diagnose real-world performance anomalies for modern desktop applications? (2) How does Argus compare to other performance debugging tools? (3) How useful are Argus’s weak edges and their optimizations in mitigating tracing inaccuracies? (4) How much overhead does Argus’s tracing tool incur? Unless otherwise indicated, all applications and tools were run on a MacBookPro12,1 with an Intel Core i7 CPU, 16 GB RAM, and an APPLE SM0512G SSD.

7.1 Diagnosis Effectiveness

We evaluated Argus on 12 real-world user-reported performance issues in 11 popular desktop applications, which we collected and reproduced, as listed in Table 2. We are especially interested in evaluating performance issues that have been hard to troubleshoot. Except for B11, all of these are open issues, meaning their root causes were previously unknown to developers. For B2, the reported issue was “fixed” in the latest version (due to refactoring or platform upgrade) but the root cause remained unknown. Nine applications, or some of their components, have source code available, whereas two applications are closed-source. Source code was used to validate whether the correct root cause was diagnosed for the performance issues, but all evaluation was performed on the released application binaries. We have also used

ID	App	Performance Issue	Age
B1	Chromium	Typing non-English in searchbox, page freezes.	7 yr
B2	TeXstudio	Modifying Bib file in other app gets pinwheel.	2 yr
B3	BiglyBT	Launching BiglyBT installer gets pinwheel.	1 yr
B4	Sequel Pro	Reconnection via ssh causes freeze.	4 yr
B5	Quiver	Pasting a section from webpage as a list freezes.	5 yr
B6	Firefox	Connection to printer takes a long time.	1 mo
B7	Firefox	Some website triggers pinwheel in the DevTool.	3 yr
B8	Alacrity	Unresponsive after a long line rendering.	6 mo
B9	Inkscape	Zoom in/out shapes causes intermittent freeze.	1 yr
B10	VLC	Quick quit after playlist click causes freeze.	7 mo
B11	QEMU	Unable to launch on macOS Catalina.	1 mo
B12	Octave	Script editing in GUI gets pinwheel.	2 yr

Table 2: Real-world performance issues in macOS applications.

Argus with proprietary applications like Microsoft Word for macOS, but without source code, we need to wait for vendors’ confirmation and responses; in our experience, vendors are reluctant to communicate issues with an external party.

Table 3 shows that Argus was able to diagnose all 12 performance issues, including all longstanding open issues. As listed in Table 4, we checked the correctness of Argus’s diagnosed root causes in three ways: (1) inspecting the corresponding source code if available, (2) dynamic patching with `lldb` based on the diagnosed root cause to fix the problem, and (3) confirmation by developers. The last one is ideal, but not always feasible; we reported our findings to developers for seven issues, but only received two responses. Only the root cause of B11 was previously known, which Argus returned correctly (Grd). For B1, B7, and B10, we validated the diagnosed root causes by analyzing the source code (Src). For B2 and B4, we received confirmation from the respective application developers that Argus correctly diagnosed the root cause for these open issues [8, 9] (Dev). For example, for B4, the Sequel Pro developers suspected a particular Cocoa Framework API does not work as expected, but could not pinpoint the exact place to fix it. Argus determined the defect was in their installed callback function, and we submitted a pull request [8] to fix the issue. B8 was fixed in an official developer patch after we reported the root cause (Fix). For the remaining issues, we confirmed the issue was resolved by dynamically patching the application based on the root cause (Dyn). We describe a few of the performance issues in further detail, but omit others due to space constraints.

B1-Chromium: This is the Chromium performance issue discussed in Section 2. Argus analyzes the trace graph, pinpoints the circular waits between renderer main thread and browser main thread with the interactions of daemon processes like `fontd`. Argus not only localizes the problematic execution segment (waiting on a condition variable), but also the sequence of events leading to this issue. The same issue occurs in Chrome. We also reported our findings to Chrome developers, but received no reply.

B2-TeXstudio: TeXstudio [55] is an IDE for creating LaTeX documents. Users reported when they modified a bibliography

ID	Root Cause Identified
B1	circular wait between renderer and browser main threads.
B2	long running function calculating line indices in document.
B3	recursive invocations of accessible objects in GUI.
B4	UI event loop mishandling input causes deadlock with ssh.
B5	paragraph value never equals last paragraph inside web view.
B6	sleep waiting on chain of daemons, the last being nsurlsessiond.
B7	excessive garbage collection on the main thread.
B8	excessive copy of rendering cells when searching potential URL.
B9	excessive memory operations for trimming and compositing.
B10	termination signal before displaying thread ready; deadlocks.
B11	window adjustment before it finishes launching; deadlocks.
B12	readline thread writing tty repeatedly, main thread waiting.

Table 3: Root causes identified by Argus.

file with another application, TeXstudio froze with a spinning pinwheel. We reproduced this case by running `touch` from a terminal on a 500 entry bibliography file, which immediately caused a spinning pinwheel to appear in *TeXstudio*'s window. Argus analyzes the trace graph and identifies five causal paths, ordered by likelihood of causality. The first path connects multiple entities: `Terminal`→`WindowServer`→`bash`→`kernel_task`→`fseventd`→`TeXstudio`—and suggests the following root cause chain. `touch` triggers a change in the file metadata. `fseventd` notifies `TeXstudio` and invokes a callback handler. `TeXstudio` executes a function `QDocument::startChunkLoading`, and causes busy processing in `TeXstudio`'s main thread. Argus also outputs the call stack with the busy APIs, `startChunkLoading` and `QDocumentPrivate::indexOf()`. We reported our findings to the developers and received confirmation that the diagnosis is correct.

B5-Quiver: *Quiver* [7] is a closed-source notebook application for mixing text, code, Markdown, LaTeX, etc. Users report that applying bullet points to a text cell without an empty line at bottom causes a spinning pinwheel [6]. Based on the Argus trace graph, there is a hanging vertex in the WebKit component used by Quiver. In particular, WebKit hangs in executing `InsertListCommand::doApply` when applying the list command to the Webview context from Quiver. The hang occurs because of an infinite loop bug in WebKit rather than Quiver. We verified the root cause by changing the comparison result of the loop with `lldb`, which enables Quiver to display the bulletin points without a spinning pinwheel. We reported our findings to the developers, but received no reply.

7.2 Comparing with Other Approaches

We compared Argus versus other state-of-the-art tools for diagnosing the performance issues in Table 2. We used two widely-used traditional debugging and profiling tools from Apple, `spindump` [10] and `Instruments` [12]. For `spindump`, we enable it once the performance issue appears, and repeat the process five times to eliminate bias on the start timing.

	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12
<code>spind.@top1</code>	X	X	X	X	X	X	X	X	X	X	✓	X
<code>spind.@top3</code>	X	X	X	X	X	X	✓	X	X	X	✓	X
<code>spind.@top5</code>	X	X	X	X	X	X	✓	✓	X	X	✓	X
<code>spind.@top10</code>	X	X	X	X	X	✓	✓	✓	✓	X	✓	X
<code>Instr.@top1</code>	X	X	X	X	X	X	X	✓	X	X	X	X
<code>Instr.@top3</code>	X	X	X	X	X	X	X	✓	X	X	X	X
<code>Instr.@top5</code>	X	X	X	X	X	X	✓	✓	X	X	X	X
<code>Instr.@top10</code>	X	X	X	X	X	✓	✓	✓	✓	X	X	X
<code>AppInsight</code>	X	X	X	X	X	X	✓	✓	X	X	X	X
<code>Panappticon</code>	X	X	X	X	✓	✓	✓	✓	X	X	X	X
<code>Argus</code>	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
<code>no weak edges</code>	X	X	✓	✓	✓	✓	✓	✓	✓	X	X	✓
<code>w/critical path</code>	X	X	✓	X	✓	✓	✓	✓	✓	X	X	X

Argus result validation Src Dev Dyn Dev Dyn Src Fix Dyn Src Grd Dyn

Table 4: Comparing Argus with other debugging tools.

`spindump` separately ranks the symbols from all sampled call stacks and only the top of call stacks. We examined the top N symbols and their corresponding call stack information. For `Instruments`, we enable its time profiler in the background when reproducing the bugs, and analyze its data from two seconds before the performance issue occurs to three seconds after. We rank APIs in the reported call trees with CPU time percentage and filter out system routines. Then, we select the top N APIs for investigation. We used values from $N = 1$ to $N = 10$. We also used two causal tracing tools, the macOS version of `Panappticon`, as discussed in Section 2, and `AppInsight` [40]. Since `AppInsight` was originally built for Windows, we reimplemented a version for macOS which captures trace events, constructs trace graphs, and follows the path analysing rules for diagnosis according to `AppInsight`'s design.

Table 4 shows the results for using the different tools, including the results for Argus discussed in Section 7.1; checks indicate correct root cause diagnosis. All of the other tools diagnosed much fewer performance issues than Argus. `spindump` diagnosed at most five issues. It captures the state near the symptom point but cannot deduce how the execution reaches a problematic point, especially in the presence of highly concurrent and asynchronous execution across different entities. `Instruments` diagnosed at most four issues. It only outputs the most costly functions, which are helpful for performance optimizations but may not be for troubleshooting specific performance issues. Neither of the causal tracing tools did any better because the constructed trace graphs are highly inaccurate. `AppInsight` only diagnosed two issues while `Panappticon` diagnosed four issues.

7.3 Mitigation of Trace Graph Inaccuracies

We evaluated the effectiveness of Argus in mitigating trace graph inaccuracies in diagnosing the performance issues in Table 2. Table 4 shows the benefits of weak edges and Beam

	Events	Vertices	Edges		
			Total	Strong	Weak
Max	12.3M	1.68M	1.62M	751.3K	864.6K
Min	260.8K	15.1K	25.5K	17.5K	8.01K
Mean	3.31M	349.5K	358.4K	188.8K	169.6K
Med	1.02M	97.3K	172.6K	111.9K	60.71K

Table 5: Argus trace graph statistics.

search. Argus diagnoses eight issues if it discards weak edges (no weak edges), and seven issues if it uses traditional critical path analysis instead of Beam search (w/critical path). In both cases, Argus still performs better than other tools.

Table 5 shows that the Argus trace graphs include hundreds of thousands to millions of events, and on average have 350K vertices and up to 1.68M vertices. Graphs are in general dense, with an average of 358K edges. A significant percentage, 40% on average, of the edges are tagged as weak edges. To avoid abusing weak edges and overwhelming the diagnosis, Argus applies the optimizations discussed in Section 5. Figure 10 shows the percentages of potential weak edges that Argus excludes from the trace graph for different techniques: call stack similarity, wait on end of task in a thread, acquire worker threads, and kernel task delegate. Call stack similarity was most effective in pruning potential weak edges.

We evaluated the sensitivity of Argus’s beam search settings: beam width, lookback steps, and penalty function coefficients a and b . Figure 9 shows the number of diagnosed issues when changing one setting and leaving the rest at their defaults. The settings for beam width and lookback steps are robust. Larger settings increase the diagnosis effectiveness, but if they are too large, the Argus debugger could run out of memory or time out for large trace graphs. Changing penalty function coefficients can significantly change the number of diagnosed issues. In general, small coefficients from two to four are better. Overall, the results indicate that Argus is practical, and developers do not need to spend much effort to tune search settings.

7.4 Performance

We measured the time to run the Argus grapher and debugger for diagnosing each of the performance issues in Table 2. Figure 8 shows the time varies for different issues, ranging from 49 s (B12) to 9870 s (B1). Constructing the trace graph is the dominant cost. Running the beam search diagnosis algorithm on the graph is fast, taking at most 144 s (B10).

We also measured the overhead of the Argus tracer using various CPU, memory, and I/O benchmarks running on a live deployment of Argus on a MacBookPro9,2 with an Intel Core i5-3210M CPU, 10 GB RAM, and a 1 TB SSD. We first measured five runs of the iBench Cocoa benchmark [35], with and without Argus, to measure overall performance. The reported scores were 6.14 with 0.027 standard error without Argus trac-

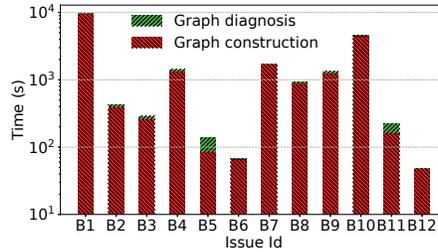


Figure 8: Argus diagnosis time.

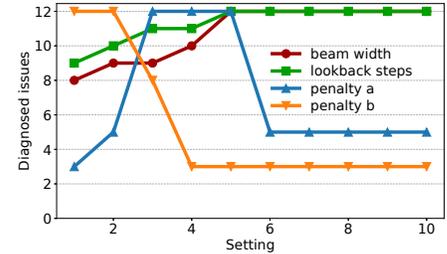


Figure 9: Sensitivity of beam search settings.

ing and 6.13 with 0.025 standard error with Argus tracing enabled. Argus only has a 0.16% performance degradation on average. In comparison, with Instruments, the reported score was 6.04, showing a 1.6% performance degradation. We next ran the Chromium Catapult benchmarks [1] to evaluate CPU performance, with and without Argus tracing. Figure 11 shows that Argus overhead is less than 5%. The average overhead for real and user time was 3.36% and 2.15%, respectively. sys overhead was higher because Argus tracing in libraries involves crossing the user-kernel boundary. Finally, we ran Bonnie++ [22] and IOzone [19] I/O benchmarks to evaluate I/O performance, with and without Argus tracing. Figure 12 shows the I/O throughput measurements. Argus tracing has almost no overhead for sequential character read and write operations and less than 10% overhead for block read and write operations.

8 Discussion and Limitations

Diagnosis in Argus may require the anomalous execution trace as well as the normal one for comparison. Obtaining the latter is not difficult. Persistent performance problems are typically eliminated before release, so the remaining issues are often non-deterministic, only occur with specific input events (e.g., typing special characters), and disappear with other events.

The quality of the Argus diagnosis results is affected by edge annotation accuracy. Beam search helps tolerate errors by inspecting multiple paths, but its settings can affect diagnosis effectiveness, as discussed in Section 6.

Argus addresses performance issues that are reflected in the underlying execution sequences and CPU time. It does not handle performance issues due to contentions among userspace threads or incorrect settings of UI elements.

Argus supports closed-source applications and libraries, but its tracing infrastructure requires slight source-level kernel modifications. System libraries such as CoreFoundation are patched at the binary level. Binary instrumentation could also be used to implement kernel changes, but is more cumbersome. Vendors of proprietary OSes have incentives to enhance their existing tracing mechanisms, and may conceivably adopt Argus kernel modifications.

We have not yet ported Argus to other OSes, but modern OSes share many similarities and provide tracing facilities that can support Argus, such as ETW [39] in Windows and LTTng in Linux [4]. Therefore, we are hopeful that our ideas are generally applicable to other OSes.

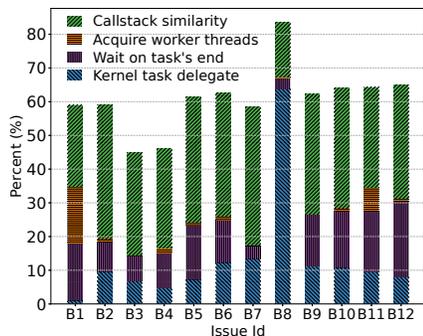


Figure 10: Potential weak edges pruned.

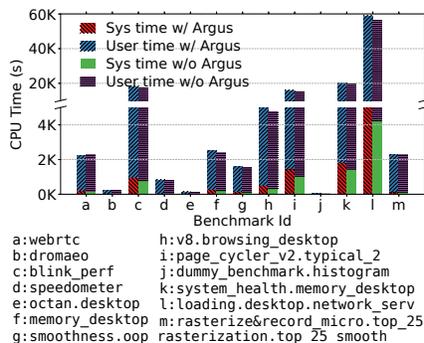


Figure 11: CPU overhead.

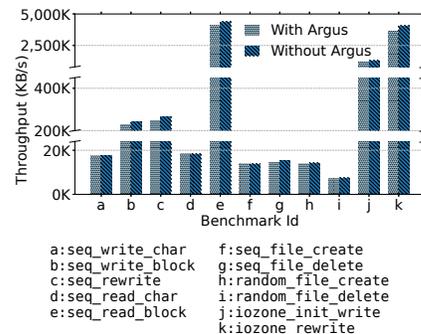


Figure 12: I/O overhead.

9 Related Work

Many causal tracing solutions have been proposed for networked and distributed systems, including Magpie [14], XTrace [27], Dapper [44], and Pivot Tracing [38]. These systems typically attach metadata to each request, propagate the metadata to all components, and stitch the traces. This approach assumes (1) the system is composed of white-box components that can be easily modified; (2) these components communicate in uniform interfaces. Neither assumption is true for desktop systems. Magpie [14] does not use metadata propagation but assumes a manual schema to extract and join events from different components’ logs. The extracted traces are limited by what each component chooses to log. However, desktop components typically are packaged as release builds that only log critical events, and logging practices among components vary greatly, which makes writing uniform schema difficult, time-consuming, and fragile.

Some causal tracing tools have been developed for mobile applications. AppInsight [40] interposes on the interface between applications and Windows Mobile frameworks and assumes that applications follow the event callback programming idiom. Panappticon [56] traces low-level events in Android and assumes two asynchronous programming idioms, message queue and thread pooling. Neither of these approaches is effective for desktop applications such as those in macOS.

Profiling or static code analysis are typically ineffective for detecting performance issues [23, 34]. Several solutions [29, 54] detect performance anomalies by leveraging logs and call stacks. Other works [21, 24, 42, 50] apply machine learning methods to identify anomalous events. Yu *et al.* [52] study the performance impact of Windows device drivers in real-world execution traces and propose to extract wait graphs from the execution traces. Several solutions [15–17] infer models from logs for distributed and concurrent systems, and use them to automate the detection of anomalous behavior when systems are exposed to new workloads and environments. These systems are orthogonal to Argus, as Argus’s goal is to diagnose an already-detected performance anomaly.

Argus is complementary to the work on concurrency bugs and race detection [18, 25, 26, 33, 36, 47, 48, 51, 53]. The former typically checks one (server) program, while Argus targets desktop applications where the defect often involves user inter-

action events, daemons, external frameworks or other applications. The latter usually focuses on testing and eliminating bugs before software is released, while Argus focuses on helping developers diagnose performance issues in the wild. Argus also addresses performance issues caused by other types of bugs.

10 Conclusions and Future Work

Argus is the first comprehensive causal tracing system to diagnose performance anomalies in complex desktop applications. We observe that although causal tracing is powerful and extensively studied in distributed systems, it is brittle when applied to desktop systems due to inherent tracing inaccuracies. Argus addresses this problem by introducing annotated trace graphs with strong and weak edges to account for these inaccuracies. Argus pairs annotated trace graphs with a novel beam search diagnosis algorithm and subgraph comparison mechanism to determine causal paths in the presence of these inaccuracies. We have implemented Argus across multiple versions of macOS and evaluated its effectiveness on complex desktop applications. Argus successfully pinpoints the root causes for 12 real-world performance issues in these applications, many of which had remained open for several years. Argus imposes less than 5% CPU overhead, making it fast enough for regular use.

We believe Argus’s strong and weak edge notions and inaccuracy-tolerant diagnosis algorithm may extend beyond the scope of desktop systems. In causal tracing of distributed systems, many solutions assume systems are perfectly instrumented, but in practice this is not the case. We plan to explore using Argus’s techniques in the context of distributed systems as an area of future work.

Acknowledgments

We thank our shepherd, Pedro Fonseca, and the anonymous reviewers for their valuable feedback. This work was supported in part by NSF grants CCF-1918400, CNS-1563555, CNS-1564055, CNS-1942794, CNS-1910133, and CCF-1918757, ONR grants N00014-16-1-2263 and N00014-17-1-2788, a JP Morgan Faculty Research Award, and a DiDi Faculty Research Award.

References

- [1] Catapult : Chromium benchmark. <https://chromium.googlesource.com/catapult>.
- [2] Chromium issue 115920: Response time can be really long with some IMEs (e.g. Pinyin IME (Apple), Sogou Pinyin IME). <https://bugs.chromium.org/p/chromium/issues/detail?id=115920>.
- [3] Kernel probes (Kprobes). <https://www.kernel.org/doc/html/latest/trace/kprobes.html>.
- [4] LTTng: Linux tracing toolkit - next generation. <https://lttng.org>.
- [5] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.
- [6] Quiver: Crash when applying bullet points on multiple lines of text. <https://github.com/HappenApps/Quiver/issues/21>.
- [7] Quiver: The programmer's notebook. <https://happenapps.com>.
- [8] Sequel-Ace fix reconnect timeout - accept SSH password after network connection reset. <https://github.com/Sequel-Ace/Sequel-Ace/pull/772>.
- [9] TeXstudio freezes when bib file is updated in the background. <https://github.com/texstudio-org/texstudio/issues/288>.
- [10] Apple. Activity monitor user guide: Run system diagnostics in activity monitor on mac. <https://support.apple.com/guide/activity-monitor/run-system-diagnostics-actmtr2225/mac>.
- [11] Apple. Cocoa fundamentals guide. <https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaFundamentals/Introduction/Introduction.html>.
- [12] Apple. Instruments overview. <https://help.apple.com/instruments/mac/current/#/dev7b09c84f5>.
- [13] Apple. trace: configure, record, and display kernel trace events. https://opensource.apple.com/source/system_cmds/system_cmds-671.10.3/trace.tproj.
- [14] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, pages 259 – 272, San Francisco, CA, USA, December 2004.
- [15] 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'14)*, page 468–479, Hyderabad, India, May 2014.
- [16] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, Arvind Krishnamurthy, and Thomas E Anderson. Mining temporal invariants from partially ordered logs. In *Workshop on Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques (SLAML'11)*, Cascais, Portugal, October 2011.
- [17] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'11)*, pages 267–277, Szeged, Hungary, September 2011.
- [18] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, page 209–224, San Diego, CA, USA, December 2008.
- [19] Don Capps, Carol Capps, Darren Sawyer, Jerry Lohr, George Dowding, Gary Little, Capps Capps, Robin Miller, Sorin Faibish, Raymond Wang, Tanmay Waghmare, Yansheng Zhang, Vernon Miller, Nick Principe, Zach Jones, Udayan Bapat, William Norcott, Isom Crawford, Kirby Collins, Al Slater, Scott Rhine, Mike Wisner, Ken Goss, Steve Landherr, Brad Smith, Mark Kelly, Alain Dr. CYR, Randy Dunlap, Mark Montague, Dan Million, Gavin Brebner, Jean-Marc Zucconi, Jeff Blomberg, Halevy. Benny, Dave Boone, Erik Habbinga, Kris Strecker, Walter Wong, Joshua Root, Fabrice Bacchella, Zhenghua Xue, Qin Li, Darren Sawyer, Vangel Bojaxhi, Ben England, Lapa, Vikentsi, and Alexey Skidanoy. IO-zone filesystem benchmark. <https://www.iozone.org/>.
- [20] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 32nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'02)*, pages 595–604, Bethesda, MD, USA, June 2002.
- [21] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design*

- and Implementation (OSDI'04), pages 231–244, San Francisco, CA, USA, December 2004.
- [22] Russell Coker. Bonnie++ benchmarking. <https://www.coker.com.au/bonnie++/>.
- [23] Charlie Curtsinger and Emery D. Berger. COZ: Finding code that counts with causal profiling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)*, pages 184–197, Monterey, CA, USA, October 2015.
- [24] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, page 1285–1298, Dallas, TX, USA, October 2017.
- [25] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys'11)*, pages 215–228, Salzburg, Austria, April 2011.
- [26] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. In *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'10)*, pages 221–230, Chicago, IL, USA, June 2010.
- [27] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)*, pages 271–284, Cambridge, MA, USA, April 2007.
- [28] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction (SIGPLAN'82)*, page 120–126, Boston, MA, USA, June 1982.
- [29] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, pages 145–155, Zurich, Switzerland, June 2012.
- [30] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, page 71–83, Cascais, Portugal, October 2011.
- [31] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, Seattle, WA, USA, July 1999.
- [32] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)*, pages 34–50, Shanghai, China, October 2017.
- [33] Oren Laadan, Nicolas Viennot, Chia-Che Tsai, Chris Blinn, Junfeng Yang, and Jason Nieh. Pervasive detection of process races in deployed systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, pages 353–367, Cascais, Portugal, October 2011.
- [34] Bozhen Liu and Jeff Huang. D4: Fast concurrency debugging with parallel differential analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*, pages 359–373, Philadelphia, PA, USA, June 2018.
- [35] Ramón Medrano Llamas. iBench: The Cocoa Benchmark. <https://ibench.sourceforge.io>.
- [36] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*, pages 329–339, Seattle, WA, USA, March 2008.
- [37] Jonathan Mace and Rodrigo Fonseca. Universal context propagation for distributed system instrumentation. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys'18)*, pages 1–18, Porto, Portugal, April 2018.
- [38] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)*, pages 378–393, Monterey, CA, USA, October 2015.
- [39] Microsoft. Event tracing for windows. <https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw->, 2002.
- [40] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. AppInsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th USENIX Symposium*

on *Operating Systems Design and Implementation (OSDI'12)*, pages 107–120, Hollywood, CA, USA, October 2012.

- [41] Patrick Reynolds, Charles Edwin Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'06)*, pages 115–128, San Jose, CA, USA, May 2006.
- [42] Ali G. Saidi, Nathan L. Binkert, Steven K. Reinhardt, and Trevor Mudge. Full-system critical path analysis. In *Proceedings of the 2008 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'08)*, pages 63–74, Austin, TX, USA, April 2008.
- [43] Raja R. Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H. Sigelman, Rodrigo Fonseca, and Gregory R. Ganger. Principled workflow-centric tracing of distributed systems. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)*, pages 401–414, Santa Clara, CA, USA, October 2016.
- [44] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspán, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, April 2010.
- [45] The LLDB Team. The LLDB Debugger. <https://lldb.lvm.org/>.
- [46] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R. Ganger. Stardust: Tracking activity in a distributed storage system. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'06/Performance'06)*, pages 3–14, Saint Malo, France, June 2006.
- [47] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, pages 369–384, Cascais, Portugal, October 2011.
- [48] Jingyue Wu, Heming Cui, and Junfeng Yang. Bypassing races in live applications with execution filters. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, pages 135–149, Vancouver, BC, Canada, October 2010.
- [49] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad Hoc synchronization considered harmful. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, pages 163–176, Vancouver, BC, Canada, October 2010.
- [50] 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 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, pages 117–132, Big Sky, MT, USA, October 2009.
- [51] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*, pages 485–502, Tucson, AZ, USA, October 2012.
- [52] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. Comprehending performance from real-world execution traces. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, pages 193–206, Salt Lake City, UT, USA, February 2014.
- [53] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, pages 221–234, Brighton United Kingdom, October 2005.
- [54] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, pages 293–306, Hollywood, CA, USA, October 2012.
- [55] Benito van der Zander, Jan Sundermeyer, Danel Braun, and Tim Hoffmann. TeXstudio: LaTeX made comfortable. <https://www.texstudio.org>.
- [56] Lide Zhang, David R. Bild, Robert P. Dick, Z. Morley Mao, and Peter Dinda. Panappticon: Event-based tracing to measure mobile application and platform performance. In *Proceedings of 2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Montreal, QC, Canada, September 2013.

aDFS: An Almost Depth-First-Search Distributed Graph-Querying System

Vasileios Trigonakis,¹ Jean-Pierre Lozi,¹ Tomáš Faltín,^{1,2} Nicholas P. Roth,³ Iraklis Psaroudakis,¹ Arnaud Delamare,¹ Vlad Haprian,¹ Călin Iorgulescu,¹ Petr Koupy,¹ Jinsoo Lee,¹ Sungpack Hong,¹ and Hassan Chafi¹

¹Oracle Labs, `firstname.lastname@oracle.com` ²Charles University
³KUNGFU.AI (work done while at Oracle Labs), `nicholas.roth@kungfu.ai`

Abstract

Graph processing is an invaluable tool for data analytics. In particular, pattern-matching queries enable flexible graph exploration and analysis, similar to what SQL provides for relational databases. Graph queries focus on following connections in the data; they are a challenging workload because even seemingly trivial queries can easily produce billions of intermediate results and irregular data access patterns.

In this paper, we introduce aDFS: A distributed graph-querying system that can process practically any query fully in memory, while maintaining bounded runtime memory consumption. To achieve this behavior, aDFS relies on (i) almost depth-first (aDFS) graph exploration with some breadth-first characteristics for performance, and (ii) non-blocking dispatching of intermediate results to remote edges. We evaluate aDFS against state-of-the-art graph-querying (Neo4J and GraphFrames for Apache Spark), graph-mining (G-Miner, Fractal, and Peregrine), as well as dataflow joins (BiGJoin), and show that aDFS significantly outperforms prior work on a diverse selection of workloads.

1 Introduction

Graph processing is a very active area of research, with a plethora of prior work focusing on classic graph algorithms [35, 36, 38, 48, 53, 58, 61, 62, 82, 85], graph mining [27, 29, 32, 34, 42, 54, 69, 74, 77], as well as graph querying [1, 3, 8, 10, 12, 18, 31, 86] and graph-query languages [4, 5, 11, 14, 17]. Graph algorithms (such as PageRank [57]) are typically used in batch computations, while graph mining is used to extract structural properties and compute cumulative statistics of a graph by exploring its subgraph structures.

Graph queries are a key tool for graph analysis, as indicated by the large number of existing systems and graph-query languages. Graph queries provide an expressive interface for interactive graph exploration with rich dynamic projection and filtering support that is analogous to SQL for relational databases (see Section 5 for further details). They focus on data connections, i.e., edges, allowing users to submit queries

with any pattern, filter, or projection. For instance, the following PGQL [14] query:

```
SELECT a1.name, a2.name, a1.country = a2.country,  
       ABS(a1.salary - a2.salary) AS salary_diff  
MATCH (a1:author)-[:likes]->(a2:author),  
       (a2)-[:likes]->(a1)  
WHERE ABS(a1.age - a2.age) <= 10  
ORDER BY salary_diff DESC
```

enumerates the authors of similar age that like each other. Answering such a query requires finding all homomorphic [45] matches of the query pattern in the target graph, while enforcing filters (e.g., `a1 IS author`) and projecting the requested output (e.g., `whether a1.country = a2.country`).

The dynamic user-defined patterns, filters, and projections, the focus on edges, and the homomorphic matching make graph query execution a challenging workload that needs to handle very large intermediate and final result sets, with a combinatorial explosion effect. For example, on the well-researched Twitter graph [47], the single-edge query `(a) -> (b)` matches the whole graph, amounting to 1.4 billion results, and the two-edge query `(a) -> (b) -> (c)` amounts to 9.3 trillion matches. This means matching the `(a) -> (b) -> (c) -> (a)` cycle needs to consider 9.3 trillion intermediate results. Compared to relational queries, graph queries can exhibit extremely irregular access patterns [51, 63] and lack of spatial locality, while calling for low-latency data access.

High-performance graph-querying systems ideally need to (i) keep the computation in main memory to guarantee low latency, (ii) scale out to multiple machines in a distributed manner to handle graphs and queries that exceed the capacity of a single machine, and (iii) control their memory usage at the machine level. Controlling memory consumption during query execution becomes paramount for cloud graph-processing services, in which multiple users submit queries that produce results of unpredictable size. Allowing a single query to monopolize memory would hinder service quality for users running other queries.

Query execution on graphs is typically based on one of the two classic graph-traversal strategies: Breadth-first search (BFS) or depth-first search (DFS). Both BFS and DFS have major advantages and drawbacks for distributed graph queries:

BFS traversals are easier to parallelize but, as with distributed joins, suffer from explosion in the size of intermediate results, cannot be easily pipelined, and stress the network bandwidth to shuffle data across levels of pattern matching. DFS traversals reduce the size of intermediate results, but are challenging to parallelize and result in random data access patterns, wasting locality when iterating over neighbors.

In this paper, we introduce aDFS (almost-DFS): A novel distributed graph-querying system that brings the best of both DFS/BFS worlds. aDFS extends the graph-processing capabilities of PGX.D [39] with queries and processes graphs partitioned across multiple machines *fully in memory*, combining BFS and DFS traversals to *bound the maximum amount of memory* required for query execution, while achieving a *high degree of parallelism*. DFS, together with a distributed flow-control mechanism, guarantee that the amount of runtime memory remains within limits, while the BFS exploration allows for better locality and parallelization during execution.

Worker threads in aDFS mainly prioritize DFS execution for completing—and thus freeing—intermediate results. The execution switches to BFS when matching a remote edge (i.e., an edge pointing to a remote machine) or when the runtime detects that the query contains limited parallelism (i.e., a small set of intermediate results). To elaborate, for local edges, worker threads perform DFS, unless aDFS detects that there is a limited amount of available work on the local machine, in which case they switch to per-thread BFS exploration until there is enough parallelism. For remote edges, threads buffer the matched intermediate results and continue with matching the next edge in a BFS manner (i.e., the next edge is possibly at the same depth as the current one). Once a buffer is full, the worker thread sends its contents to the target machine, unless it is blocked by the flow-control mechanism, which enforces target memory limits. Section 3 expands on the design and implementation of aDFS.

Section 4 thoroughly evaluates aDFS and shows that it is capable of executing trillion-scale queries, with a 10GB per-machine runtime memory cap. When running our largest query, aDFS computes a 9.3 trillion count pattern on the Twitter graph with a rate of 7.3 billion matches per second. We compare aDFS to two graph systems (i.e., Apache Spark GraphFrames [31] and Neo4j [10]) and two relational databases (i.e., MonetDB [9] and PostgreSQL [15]) using the LDBC graph and query suite [68]. aDFS completes the set of queries 43 and 53 times faster than GraphFrames and Neo4j,¹ respectively, and 8 and 26 times faster than MonetDB and PostgreSQL, respectively (as Section 4.2 shows, LDBC is “relational-friendly”). We also compare aDFS to these four systems with schema-less graphs and show that either aDFS is 16 to 9,200 times faster than the rest, or the other systems simply fail to complete the queries. Finally, we compare aDFS with (i) three state-of-the-art graph-mining systems: G-Miner [27], Fractal [32] and Peregrine [42], as well as (ii) BiGJoin [19], a dataflow join system. We show

that aDFS is up to 12, 625, and 18 times faster than G-Miner, Fractal and Peregrine, respectively, and performs comparably to BiGJoin on mining-oriented workloads. We discuss related work further in Section 5.

The main contributions of this paper are the following:

- aDFS, which is, to the best of our knowledge, the first graph-querying system that strictly bounds runtime memory while operating with fully-distributed computations over partitioned graphs;
- The novel combination of DFS (for eager completion of intermediate results), BFS (for performance), and flow control (for controlling the size of the intermediate state) to achieve performance and scalability while capping memory usage; and
- The evaluation of aDFS, which shows that aDFS significantly outperforms the state of the art and is capable of executing queries with trillions of matches.

2 Background and Motivation

Representing data as a graph is becoming increasingly popular. The main advantage of graphs is that they focus on modeling fine-grained relationships between entities. In contrast, the relational model concentrates more on rows and relies on the heavyweight primary-key foreign-key (PK-FK) and join mechanisms to link entities. However, when using graphs, different models, data representations, and ways of exploring them have a major effect on processing performance.

2.1 The Property Graph (PG) Model

Property graphs represent the graph topology as vertices and edges, and store *properties* and *labels* separately. Properties can be associated to any vertex or edge and take the form of typed key-value pairs. Labels are key-only and represent types or categories, e.g., *person* or *animal*. Separating the topology from properties avoids the proliferation of edges and allows for quick traversals of the graph over its real structure.

2.2 Graph Pattern-Matching Queries

Several languages for graph querying exist, such as PGQL [14], SPARQL [17], Gremlin [5], and Cypher [11]. In its simplest form, graph querying makes it possible to find patterns in graphs, with filters and projections. aDFS uses PGQL, which is modeled after SQL: Projection and aggregation operations are the same as in SQL, including `GROUP BY` and `ORDER BY`. PGQL adds support for graph patterns and vertex and edge labels. For example, the query presented in Section 1 adds the `MATCH` clause to an otherwise valid SQL query. It matches patterns that are *homomorphic* [45] to the `(a1) -> (a2) -> (a1)` cycle while enforcing filters (e.g., `a1` has label `author`), and it projects or aggregates the requested data—including even arbitrary expressions—out of the matched vertices and edges.

Graph queries require homomorphic matching of the pattern, as data is projected out of all matches, even if they are permutations of each other. In contrast, graph-mining systems

¹Using Neo4j Community Edition (benchmarks not audited by Neo4j).

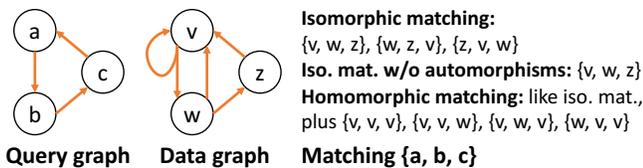


Figure 1: Homomorphic vs. isomorphic matching of a pattern.

focus on *isomorphic matching* [34] and often use automorphism elimination to prune the search space further. Figure 1 highlights the differences between isomorphic and homomorphic matching of a query pattern. In graph queries, isomorphic matching and automorphism elimination can be simulated with filters and/or with query specifiers, such as `GROUP BY`.

In this work, we focus on the backbone of graph pattern-matching queries. Accordingly, aDFS only supports a subset of PGQL 1.1 [14]; in particular, two following features are missing: Regular path expressions and subqueries. Nevertheless, we design aDFS with these features in mind and we intend to work on them in future work.

2.3 Graphs vs. Relational Joins and RDF

In PG graph systems, edges are stored explicitly and can be traversed directly. In contrast, in relational databases, relationships are represented with PK-FK. Following any relationship means joining two tables—or doing a self-join if the keys belong to the same table—and producing the intermediate result. Therefore, while matching multiple-hop paths is a relatively cheap operation in graph systems, doing the same in SQL requires a chain of multiple expensive join operations that materialize intermediate results. Thus, graph systems can be much more efficient than relational databases when it comes to matching graph patterns (see Section 4 for a comparison).

Another alternative model is the Resource Description Framework (RDF) that uses $\{subject, predicate, object\}$ triples to represent graphs, which can be queried with languages such as SPARQL [17]. RDF became popular with the semantic web [21] and has been the model of choice for many graph databases starting in the early 2000s [37, 80]. A number of works have focused on distributed RDF graphs [40, 64].

Although the RDF model is equivalent to PG in terms of expressiveness, there are differences: (i) RDF adds links for every graph data piece, including constant literals, (ii) it does not have explicit vertices/edges—yet it can be viewed as representing graphs, and (iii) it does not store properties separately. The de-facto implementation of RDF triples results in similar PK-FK behavior as aforementioned for relational databases. Triples force RDF systems to process and join a much larger number of intermediate results using e.g., a key-value-style storage, and lose the graph structure, resulting in slower neighbor lookup. To address these drawbacks, some RDF systems use asynchronous processing [37], or compute graph indices, using e.g., the CSR representation, to mock the graph structure [80]. aDFS focuses on PGQL queries and the

PG model, avoiding complex and expensive joins. We note that the pattern-matching part of query execution is largely orthogonal to the graph model and aDFS’s techniques could be used for RDF graphs.

2.4 DFS, BFS, and Intersections for Graph Exploration

DFS can expand one intermediate result at a time, starting from the first variable in the pattern and continuing to the next ones until the whole pattern is matched. However, this behavior results in totally random accesses and is impractical for distributed graph traversals: The only way to continue with strict DFS is to directly send the intermediate result to the remote machine and wait until it is picked up and completed.

Thus, graph exploration is traditionally done using BFS: For each query edge (hop), the entire result set is computed, and only then does the exploration of the next hop start. This approach has two main advantages: (i) it is easy to implement, as work is naturally divided into simple steps (hops), and (ii) it is relatively easy to parallelize, as the entire input is known before processing a hop (of course, skewed vertex degrees still pose a problem). However, BFS has one major shortcoming: Because the intermediate result set is produced between stages, an intermediate result-set explosion can quickly occur.

Figure 2 illustrates this issue showing the average total per-machine memory usage and execution time when matching cycles of various lengths using aDFS and BFS (implemented in our runtime) on a small graph [16] (875K vertices and 5.1M edges). While both approaches are able to match cycles of length one to four with similar performance, the memory consumption of BFS explodes for five-hop cycles at approximately 60GB on each of the eight machines in the experiment, and BFS crashes with six-hop cycles after 96 minutes when one machine runs out of memory (~768GB). Meanwhile, the memory consumption of aDFS is almost constant.

Recent graph-mining and graph-querying systems [22, 42] adopt a pattern-matching approach that relies on intersecting neighborhood lists. Instead of being vertex-centric (i.e., starting from vertices and following edges), the intersection-based approach focuses on edges. The benefit of the intersection-based model is that it takes $O(|V|)$ steps since it allows intersecting multiple incoming edges at a time, as compared to the vertex-based approaches that are $O(|E|)$. However, intersections require complete subgraph parts to operate. This necessitates pulling/gathering possibly large amounts of data from remote machines. To make things worse, queries enumerate all automorphisms (i.e., the exploration space could locally explode) and offer arbitrary user filters and projections, meaning that in an intersection-based model, one would need to pull not only the vertex/edge data, but also all the properties required by the query. Therefore, we use a vertex-centric approach in aDFS that builds mini-frontiers based on the first query vertex and enables aDFS to operate on fully partitioned graphs with limited memory.

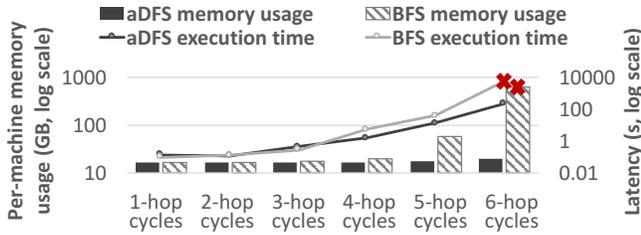


Figure 2: Matching cycles using aDFS vs. BFS.

3 aDFS: A Pattern Matching and Querying System for Distributed Graphs

The main design goals of aDFS are (i) enabling fast, fully in-memory distributed queries of any size, while (ii) allowing for limited, controllable memory consumption during execution. The rationale for these two goals is as follows. First, high-performance graph queries demand in-memory execution and the ever-increasing size of data calls for distribution. Second, server systems, especially in cloud deployments, are shared by multiple concurrent users, hence no single query can be permitted to saturate the system memory. aDFS achieves these two goals through the following design principles.

1. **§3.3: DFS-first and asynchronous communication.**

The eager match completion of DFS gives aDFS fine-grained control on the size of intermediate results during query execution, but strict DFS would be inefficient when matching a remote edge, i.e., an edge that leads to a remote machine. For that reason, worker threads do not block when encountering a remote edge, but place the intermediate result in a message buffer and continue with other local work instead. Buffers batch intermediate results: once full, a buffer’s contents are asynchronously sent to the remote machine for further processing. Threads only need to block if flow control dictates so. This buffering results in essentially BFS exploration of the remote edges of a vertex.

2. **§3.4: Flow control.** Cross-machine communication is controlled through a flow-control mechanism that caps the number of in-flight intermediate result buffers. The finite nature of these message buffers allows strictly configuring the amount of runtime memory that aDFS requires, while the flow-control mechanism guarantees query termination and deadlock freedom.

3. **§3.5: Dynamic local DFS/BFS.** Besides the BFS style of buffering for remote edges, aDFS includes a dynamic approach for deciding whether to go DFS or expand with BFS for local matches in order to improve parallelism, locality, and work sharing across threads.

Before diving into these design principles, we first present the architecture of aDFS from a high-level point of view (Section 3.1) and describe how aDFS generates execution plans for graph queries (Section 3.2).

3.1 High-Level aDFS Architecture

Figure 4 shows the high-level architecture of aDFS. Graphs are kept in memory and are partitioned across machines based on simple random vertex partitioning. Random partitioning achieves cross-machine balance and does not overfit to the workload. Of course, intelligent partitioning schemes could bring performance benefits and are left for future work. aDFS’s approach is orthogonal to partitioning, i.e., it can work with any partitioning approach.

For efficient traversals, graphs are stored in the classic CSR (Compressed Sparse Row) graph format. Due to graph partitioning, messaging is necessary for moving intermediate results to the machine which holds the target vertex. aDFS maintains two threads on dedicated cores on each machine for messaging; a sender and a receiver. Consequently, worker threads in aDFS place their messages in software queues, from where they are picked up by the sender.

3.2 Distributed Query Execution Planning

Users submit declarative PGQL queries [14] to aDFS. As Figure 3 illustrates, each query goes through three transformation steps (marked i through iii) before being executed in step iv.

Step i: Logical query planner. The first step translates the PGQL query into a logical query plan, which consists of the logical operators of Table 1. Similar to relational query planning, a given query can be executed by multiple logical query plans. In the example of Figure 3, an alternative plan could rewrite the query as $(a) - [e] - (c) \rightarrow (a) \rightarrow (b)$. This first step directly translates the query to an admissible plan, which is then optimized in the following steps.

Step ii: Distributed query planner and optimizer. This step specializes the logical query plan by taking into account the specific characteristics of aDFS’s runtime. The query planner rewrites the logical plan in terms of *stages* and transitions from one stage to another (called *hops*). A stage is responsible for matching or accessing exactly one vertex and contains all the information necessary for matching the corresponding vertex and for transitioning to the next vertex with a hop. In the example of Figure 3, the topmost stage “a” matches the first vertex a of the query, while the next one matches b. An out-neighbor hop takes the execution from a to b.

aDFS supports four types of hops that specialize for distributed execution: *neighbor match*, *edge match*, *output*, and *inspection*. Neighbor and edge hops have the same behavior as the corresponding logical operators in Table 1. An *output* hop produces a final match using the current intermediate result and is always used in the last stage of a match.

Inspection hops are specific to distributed processing: They bring the current intermediate result back to an already matched vertex in order to continue query evaluation. In the example of Figure 3, after matching a and b of $(a) \rightarrow (b)$, the query again needs the neighbor list of the already matched vertex a in order to continue with matching $(a) \leftarrow (c)$. Since the matched vertex b might be in a different machine than a,

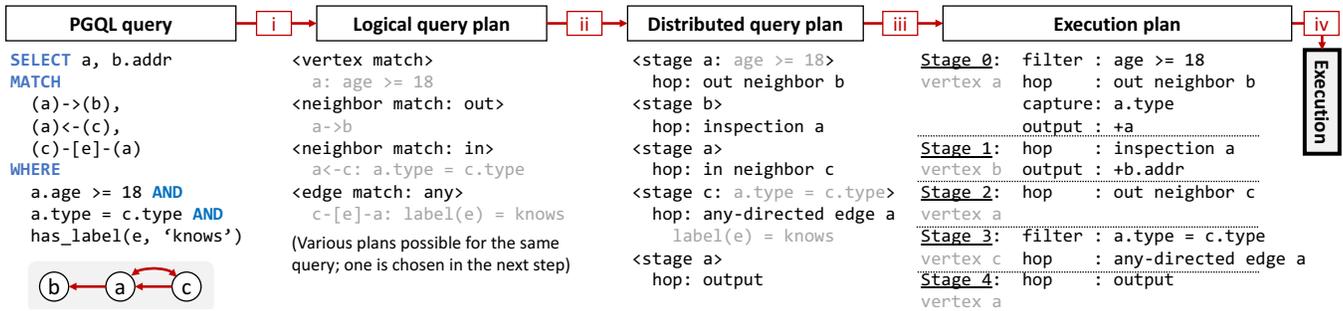


Figure 3: From a PGQL query to aDFS execution. Three transformation steps before execution.

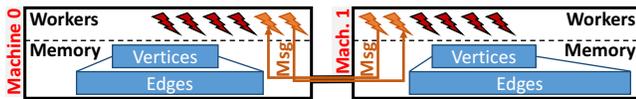


Figure 4: High-level architecture of aDFS.

the query planner introduces an inspection step to “link” this disconnected pattern and bring back the context to the machine of a . If a resides in the current machine, an inspection hop is essentially a no-op.

In this step, aDFS rewrites the logical query plan with a cost-based optimizer, implemented using dynamic programming, that is based on the following heuristics: (i) heavily filtered vertices are preferred for the earlier stages of the plan, (ii) inspection hops are not free and increase the plan’s cost, and (iii) the cost of an edge hop is approximately \log of the cost of a neighbor hop, as it can be implemented with a binary search in the neighbor list of the source vertex. The optimizer further detects whether a query has a single starting vertex, by extracting ID equality filters (e.g., $ID(\text{person}) = 123$). In the example of Figure 3, the optimizer rewrites the query as $(a) - [e] - (c) \rightarrow (a) \rightarrow (b)$ because it avoids an inspection hop and a and e are more filtered as compared to b .

Steps iii–iv: Execution plan and execution. Finally, aDFS generates a concrete execution plan. Apart from stages and hops, the execution plan contains filters (on vertices and edges), as well as information on what data should be included in the intermediate results in order to execute filters of later stages and produce the final output. For example, in the query of Figure 3, Stage 0 must collect $a.type$, since it is

Op.	Example	Short description
Vertex match	(x)	Match vertices of the graph (without following any edge)
Neighbor match	$(x) \rightarrow (y)$	Having matched the left vertex x , match its neighbors y ; can be in-, out-, or any-directional
Edge match	$(x) \rightarrow \dots$ $(y) \rightarrow (x)$	The vertex x is known (already visited)—test whether x exists in the neighbor list of the left vertex y ; can be in-, out-, or any-directional

Table 1: Graph operators used in the logical query plan.

required by the filter of Stage 3. Similarly, Stage 0 must put vertex a in the intermediate result as it is part of the projection of the query. Overall, each stage builds up the intermediate result such that another thread, local or remote, can pick it up and continue the computation. The resulting execution plan is then submitted to the aDFS runtime, on which we focus next.

3.3 aDFS’s Depth-First Runtime

The runtime of aDFS is based on the *stage* and *hop* constructs described above. aDFS initiates query execution by applying Stage 0 (matching of the first vertex variable of the execution plan) to each vertex of the graph. This bootstrapping process happens (i) across machines, i.e., each machine starts from the locally-stored vertices, and (ii) concurrently within each machine, i.e., each worker thread handles a distinct set of vertices and performs the bootstrapping process on these vertices one after the other. Hops that follow remote edges send the intermediate match (batched) to the destination remote machine where they are picked up and taken over by a local thread.

Bootstrapping a match. Figure 5 includes a high-level activity diagram of the aDFS runtime. Completing the execution of this diagram from Stage 0 to the last query stage implements the complete matching starting from a single vertex of the graph. We explain these steps using the example of Figure 6. Text in the *blue italic face* represents the activities in Figure 5. The aDFS runtime assigns vertex *Joe* (the dark gray rounded rectangle of Figure 6) to a worker thread t , which tries to generate new matches. The thread first tries to match *Joe* with Stage 0’s $p1$ using *apply stage*. If the filter $p1.name = \text{"Joe"}$ returned `FALSE`, the thread would try to *backtrack* to a previous stage and, because there is none, it would simply complete this invocation. If there were more top-level vertices to explore, t would start again with a different vertex.

In the example of Figure 6, we assume that the execution plan matches vertex $p1$ as Stage 0. $p1$ matches *Joe* and t continues with the *hop: follow next edge* operation, starting from edge ①. Since the `:friend` label filter is satisfied and the edge is local, t proceeds via *DFS next stage* to Stage 1 where $p2$ is matched with the vertex that has `age = 20`. At this point, since the filter $p2.age < 35$ is satisfied and there is no next stage, t produces a query output row and *backtracks*

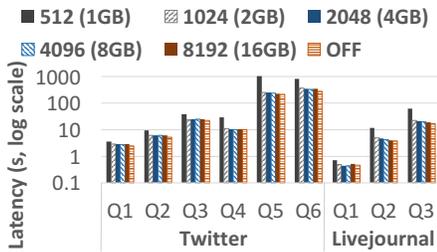


Figure 7: Performance of simple queries (8 machines) with different flow-control limits. In parentheses: Total per-machine maximum memory consumption.

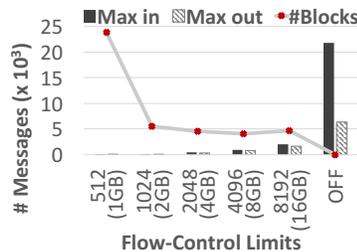


Figure 8: Messaging and blocking statistics on Q3/Livejournal with different flow-control limits. In parentheses: Total per-machine max. memory consumption.

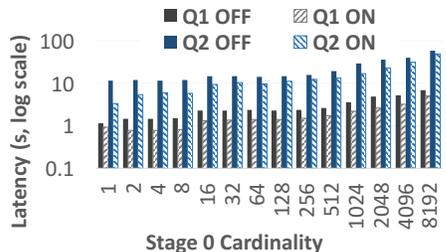


Figure 9: aDFS with dynamic local-edge BFS “ON” or “OFF” for two queries while varying the number of intermediate results of the first query stage.

amount of blocking is very high, which penalizes performance (more than $3\times$ higher latency). Still, the overhead for switching stages due to flow control is generally low: Setting N to 8,192 results in only $\sim 10\%$ performance loss as compared to no flow control (OFF), while reaching $10\times$ fewer maximum incoming messages (2,087 vs. 21,793) and $4\times$ fewer outgoing messages (1,636 vs. 6,430).

3.5 Dynamic BFS for Local Edges in aDFS

For remote edges, aDFS essentially does (per-thread) BFS: A thread matching a remote edge simply buffers the intermediate result and continues exploring and matching the same stage, which might produce new intermediate results.

While local processing could happen in pure DFS, doing so can result in artificially limited parallelism for queries that produce small sets of intermediate results. A characteristic example is queries with a very narrow starting Stage 0, such as `MATCH (a)->... WHERE ID(a) = X`; this narrow-start behavior appears in several real-life queries (e.g., the LDBC queries of Section 4). In such a query, the whole Stage 0 might produce a single intermediate result, giving limited opportunities for parallelism. For these workloads, DFS can significantly delay the expansion of intermediate results that are produced in the system (both locally and through messages).

In aDFS, we solve this DFS limitation by dynamically switching from depth-first exploration to per-thread breadth-first for local edges. aDFS maintains per-stage counts of the number of buffers with intermediate results that are ready to be taken care of by worker threads. A low number of intermediate results means that the stage has not expanded enough, hence some threads could end up not having sufficient work to perform. When threads in aDFS are processing a local edge, they use this information to decide whether to go for BFS, i.e., buffer the intermediate result in a local buffer and continue at the same stage.

In practice, we keep these local buffers small, i.e., up to a few kilobytes, in order to promote quick local work creation. We further use a *DFS threshold* to decide when to work depth-first: When the sum of the number of local buffers (produced by the breadth-first expansion) plus the number

of message buffers from remote machines is greater than $4\times$ the number of threads, threads switch to DFS. Having a low threshold plus small local buffers allows aDFS to keep the maximum additional memory consumption limited: If the DFS threshold is set to n , the maximum number of threads is t , the size of local buffers is b , and the query contains s stages, the maximum additional memory in a machine is $(n+t) * (s-1) * b$. In the configuration used for our experiments ($t = 28$, $n = 4t = 128$, $s \leq 11$, and $b = 8,192$), local buffers consume less than 12MB additional memory.

Controlled Experiment. Figure 9 illustrates the benefits of this local-match BFS mode on 8 machines (see Section 4 for detailed experimental settings) with the following two queries:

```
1: (a)->()->() WHERE ID(a) < $i
2: (a)->()-[e]->()->() WHERE e.cost < 0.5
   AND ID(a) < $i
```

using the Twitter graph extended with a uniform random edge property with values in $[0.0, 100.0)$. In both queries, the `ID(a) < $i` filter determines the cardinality of the first query stage and is used to narrow the starting point. In Q2, the edge filter also guarantees that the third stage includes a small number of intermediate results. The dynamicity of aDFS brings significant performance benefits, especially for queries with very narrow starting points. For example, for Q1 with $\$i = 1$, Machine 0 hosts the match for Stage 0; without the breadth-first mode (“OFF”), a single thread handles all the 55K local edges which lead to Stage 1. In contrast, enabling dynamic local BFS (“ON”) generates more work early on and allows splitting the work among local threads, each of which operates on approximately 2,000 vertices for Stage 1.

LDBC Q20. We also briefly analyze the BFS-mode benefits on LDBC Q20 (see Section 4 for more details):

```
MATCH (tC:tagClass) <-[:subClassOf]-(:tagClass)
  <-[:hasType]-(:tag) <-[:hasTag]-(:post|comment)
  WHERE tC.name IN ('Politics', 'Art', 'Country')
```

In this query, the first two stages match `tagClasses` and Stage 0 results in only three intermediate results due to the filter. The local BFS optimization brings 32% latency benefits (8 vs. 5.5 seconds), by better parallelizing the work across threads. Without the optimization, the most busy thread, i.e., the one that “gets stuck” in performing local DFS work the

most, spends 4 seconds in these local explorations: It matches about 1,000 vertices in Stage 1, which result in 5.2 million local matches in Stage 2 and 5 million in Stage 3. In comparison, with the optimization, the most busy thread spends only 1.6 seconds in DFS work: It handles 4 million local edges in Stage 2, which it successfully distributes to other threads with approximately 500 local BFS buffers. Overall, enabling dynamic local BFS provides significant speedup on realistic workloads, while incurring at most a 5% slowdown.

4 Evaluation

The goals of our evaluation are (i) to understand how well aDFS performs as compared to other systems (graph, relational, mining and dataflow join systems) that could be used in similar use cases, (ii) explain how different parts of aDFS contribute to performance and memory, and (iii) show how aDFS scales as we increase the number of machines.

4.1 Experimental Settings

Hardware details. We use a cluster of eight nodes, each having two Intel Xeon E-2690 v4 2.60GHz CPUs with 14 cores (hyperthreads disabled/DVFS enabled), for 28 cores in total. Each processor contains 756GB of DDR4-2400 memory and LSI MegaRAID SAS-3 3108 storage. Each node includes a Mellanox Connect-X InfiniBand card, all connected to an EDR 100Gbit/s InfiniBand network.

Graphs and queries. Unless specified otherwise, our experiments use the five graphs of Table 2. As we mention in Section 2, the scope of this paper covers user-provided fixed-pattern queries, thus aDFS implements only a subset of PGQL 1.1. Accordingly, we use the 12 LDBC Business Intelligence (BI) standard queries [68] supported by PGQL 1.1 (later PGQL versions support the remaining LDBC queries). Out of these 12 queries, four represent simple path patterns (i.e., Q4, Q17, Q23, Q24) and are directly supported in aDFS. The remaining ones either include regular path queries (e.g., ... MATCH (a)-[:knows*]->(b)), or include sub-queries in projection or filters (e.g., SELECT ... FROM (SELECT ...) ...). We devise a simplified variant of these queries in order to support the benchmark specification as closely as possible. For example, the original Q6 is:

```
SELECT id(person),
SUM((SELECT COUNT(*) MATCH (m)<-[:replyOf]-(:cmt)) AS rN
SUM((SELECT COUNT(*) MATCH (:prsn)-[:likes]->(m)) AS lN,
COUNT(*) AS msgN
MATCH (tag:tag) <-[:hasTag]- (m:post|comment)
-[:hasCreator]-> (person:prsn)
WHERE tag.name = ?
GROUP BY person, tag
ORDER BY msgN + (2 * rN) + (10 * lN) DESC, id(person)
```

We simplify the query by removing the two COUNT subqueries in projections and from ORDER BY. We plan to extend the PGQL support of aDFS in future work.

Note that the queries include patterns of varying complexity, e.g., the one in Q6 above is rather simple, while Q17 matches the following complex pattern:

```
(x:person)-[:livesIn]->(c1:city)-[:partOf]->(cy:country),
(y:person)-[:livesIn]->(c2:city)-[:partOf]->(cy),
```

```
(z:person)-[:livesIn]->(c3:city)-[:partOf]->(cy),
(x)-[:knows]->(y)-[:knows]->(z)
```

Methodology. We perform 15 runs of each query and report the median latency (in Figure 10, the error bars cover all runs). For each experiment set, we execute the queries in a per-graph round-robin fashion in order to reduce caching effects (e.g., data in the LLC or instruction caches). We use eight machines for aDFS, GraphFrames, G-Miner, Fractal as well as BiGJoin, and make sure all systems are configured to use InfiniBand. The four other systems are single machine.

Engines and their configurations. We configure aDFS to use up to 4,096 messaging buffers of 256KB per machine for messaging. This setting translates to approximately 1GB of intermediate results that can be produced per machine and limits the worst-case maximum memory consumption of a single machine to approximately 8GB (1GB outgoing, plus 7GB incoming). We further use the configuration of Section 3.5 for the local-edge dynamic BFS, resulting in up to a few MBs of extra memory per machine. Altogether, the aDFS runtime consumes approximately 10GB per machine. Of course, the graph (that resides in memory) and the final query results consume extra memory than these 10GB. We use such a low-memory configuration because (i) aDFS is designed for server deployments and we want to evaluate the performance at a realistic setting, where a single query cannot monopolize memory, and (ii) as we show in Figure 7, this configuration is already sufficient for aDFS to perform well.

We first compare aDFS to two graph systems and two relational systems which we describe below. In Section 4.5, we further compare aDFS to three graph-mining systems and a dataflow join system.

GraphFrames [31] is a distributed graph-querying system built on top of Apache Spark [2, 79]; we use version 0.7 on top of Spark 2.4.1 with 600GB executor memory per machine. **Neo4j** [10] is a single-machine graph database, which stores its data on disk but uses an in-memory cache for performance (caching effects are obvious in the first run of each query). We use Neo4j Community Edition 3.5.3 and allow it to manage the full memory of the machine. **MonetDB** [9, 26] is an in-memory column-store relational database. Its distributed support is rather rudimentary, resulting in worse than single-machine performance for our join-heavy workloads. Therefore, we use MonetDB 11.31.13 on a single machine, configured to use the whole 756GB of memory. **PostgreSQL** [15] is a relational database. We use version 11.2, tuned for a single connection with memory cache size of 564GB and 198GB of

Graph	#V	#E	Schema	Description
Livejournal [20]	484K	68.9M	No	Users and friendships
URandom	100M	1B	No	Uniform random edges
Twitter [47]	42.6M	1.47B	No	Tweets and followers
LDBC(100) [68]	283M	1.78B	Yes	LDBC social
Webgraph-UK [25]	77.7M	2.97B	No	2006 .uk domains

Table 2: The set of graphs we use in the evaluation.

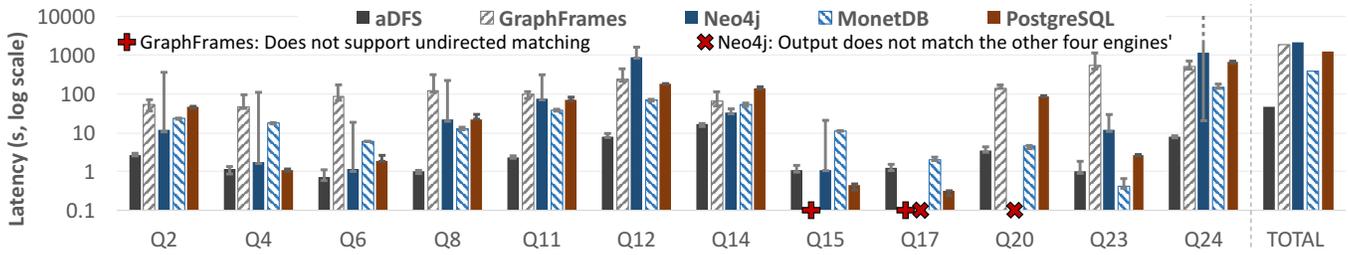


Figure 10: aDFS vs. other graph/relational systems on LDBC. Q_N is the N th LDBC BI query. Error bars show min/max latencies. TOTAL is the sum of all latencies—i.e., the time to complete a single run of all 12 (10 for GraphFrames and Neo4j) queries.

shared buffers. For both MonetDB and PostgreSQL, we use the optimized schema/indices designed for the original LDBC evaluation paper [68]. We choose these four systems as they cover a broad spectrum of data processing: Distributed graph dataframes, single-machine graph databases, and in-memory or traditional relational databases.

4.2 aDFS vs. Other Engines: LDBC

Experiment. We perform an end-to-end comparison of aDFS to the four aforementioned systems. We use the LDBC graph and BI queries which constitute an unfavorable workload for aDFS and GraphFrames: the LDBC graph has a relational schema, carefully partitioned in tables, such as `person` and `post`. For relational databases (as well as Neo4j), this schema enables the exploration of small parts of the graph for most queries. For example, the pattern `(:post)-[:hasCreator]->(:person)` (taken from an actual query) needs to only access the tables `post` and `person`, which are a relatively small part for the graph. In contrast, aDFS and GraphFrames operate on the original graph model, where the whole dataset is a single graph. The end result is that these two systems perform more broad exploration even on queries that are very narrow in terms of schema accesses.

Optimizing for relational schemas is outside of the scope of this work. Still, we choose LDBC with BI queries for our end-to-end comparison as it shows how aDFS performs on queries that can be expressed well both in graph and relational systems. The next sections focus on schema-less graphs.

Results. Figure 10 depicts the query latencies of the five systems. For most queries, aDFS is one to two orders of magnitude faster than GraphFrames. aDFS delivers $102\times$ average speedup and takes $43\times$ less total time than GraphFrames to complete the 10 out of 12 supported queries. GraphFrames translates graph queries into dataframe joins, offered by Apache Spark, which are significantly slower than aDFS’s graph traversals. Additionally, GraphFrames is memory hungry, consuming hundreds of gigabytes of memory in comparison to the small footprint of aDFS. Furthermore, aDFS completes the 10 supported queries $53\times$ faster than Neo4j, with a $35\times$ average speedup, even though Neo4j leverages the graph schema, as well as the large amount of available memory with its graph cache. With Neo4j, the whole graph

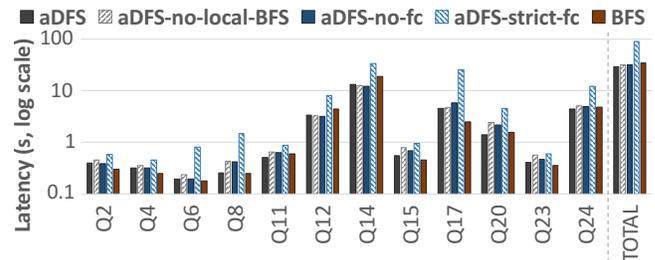


Figure 11: aDFS with various configurations on LDBC.

resides in memory after the first run: the error bars clearly show the effects of the first slow run.

Comparing aDFS to the relational systems, MonetDB and PostgreSQL, shows two different behaviors depending on the query size. For large queries, such as Q12 and Q24, which expand to large parts of the graph with long paths, aDFS is significantly faster. On the contrary, for small, very relational queries, such as Q15, Q17, and Q23, the relational systems can be faster than aDFS. This is expected given that just the distributed bootstrapping and coordination overheads in aDFS account for several tens of milliseconds. These different queries highlight the tradeoff between the relational table-focused joins and the graph exploration approach of aDFS. Overall, aDFS completes the whole set of queries 8.4 and 26 times faster than MonetDB and PostgreSQL, respectively. The average speedups are $10\times$ and $25\times$ against MonetDB and PostgreSQL, respectively. Conversely, MonetDB is $2.4\times$ faster than aDFS on Q23, while PostgreSQL is on average $2.6\times$ faster for Q4, Q15, and Q17.

In conclusion, aDFS achieves better overall performance than the four other systems while consuming lower/capped runtime memory.

4.3 Dissecting aDFS with LDBC

Experiment. We again use the LDBC benchmark to show how different design characteristics of aDFS contribute to performance and memory usage. In particular, we compare the pattern-matching-only latency of the default *aDFS* (as used in Section 4.2) to *aDFS-no-local-BFS* (we disable the machine-local dynamic BFS), *aDFS-no-fc* (we further disable flow

control), *aDFS-strict-fc* (we make flow control very strict), and *BFS* (we use the BFS implementation of Section 2.4).

Results. Figure 11 includes the results for these configurations. All in all, aDFS is the fastest. With the “dynamic BFS for local edges” option, aDFS is 31% faster on average than *aDFS-no-local-BFS* for 10 queries, while incurring 4% overhead for the remaining two queries. As we described earlier, queries often have some very “narrow” execution points with a handful of intermediate results, which leads to poor parallelization with strict local DFS. In terms of memory consumption, aDFS consumes slightly more memory than *aDFS-no-local-BFS*, not only due to the local buffers, but also thanks to better parallelization, which results in more parallel message traffic.

Disabling flow control on top of *aDFS-no-local-BFS* can bring some benefits as shown by *aDFS-no-fc*. However, the performance gains are low, as *aDFS-no-fc* hits almost no flow-control limits for this workload—i.e., local DFS and prioritizing messaging buffers from later stages of the queries result in an efficient execution flow, since none of the stages “explodes” in terms of memory. Still, *aDFS-no-fc* exhibits a 20% speedup with up to 5× higher memory consumption.

aDFS-strict-fc represents the closest realistic configuration to DFS. Processing one intermediate result at a time would naturally perform poorly, hence, we instead disable dynamic local BFS and configure each stage to have exactly one outgoing buffer to the next stage per target machine. The results show that excessive flow control reduces performance. In particular, *aDFS-strict-fc* is up to 6× slower than aDFS, while consuming up to 4× less memory.

Finally, as a reference, *BFS* implements a basic BFS-only runtime. As expected, *BFS* performs better than aDFS for certain queries, as it better leverages locality and parallelization. Still, aDFS executes the total workload 16% faster than *BFS*, while *BFS* consumes up to 6× more memory.

In conclusion, aDFS includes a set of design characteristics that when put together achieve great performance with low and controlled memory consumption.

4.4 aDFS vs. Other Engines: Large Schema-Less Queries

Experiment. The classic property graph model is schema-less, which enables users to easily query the whole dataset (unlike the relational model which requires several joins and unions of results). Therefore, we now compare aDFS to the other four systems with the schema-less graphs of Table 2: this workload shows the full power of aDFS in handling very large queries. For the relational systems, the graphs consist of two tables: One for vertices and another one for edges. Regarding queries, we use two simple patterns, a cycle $(a) \rightarrow (b) \rightarrow (a)$ as Q1 and a two-hop path $(a) \rightarrow (b) \rightarrow (c)$ as Q2, combined with aggregations in the SELECT clause (variant “a” performs a COUNT(*) and variant “b” AVG aggregations on a random vertex property). The conclusions remain the same for other patterns

and projections (not shown). Note that it is impossible to evaluate more elaborate patterns, as the competing systems can barely handle the simple patterns that we use.

Results. Figure 12 depicts the results. In most cases, aDFS is about 2 orders of magnitude faster than the other systems. For the large queries and graphs, we also see that the other systems are either not able to complete the queries within eight hours, or crash. In particular, GraphFrames crashes after having consumed its 600GB of executor memory.

The speedups of aDFS over the other systems (for the completed queries where there is no timeout) are: 16 to 62× for GraphFrames, 1,105 to 9,200× for Neo4j, 20 to 169× for MonetDB, and 60 to 190× for PostgreSQL. Neither the join-based systems (GraphFrames, MonetDB, and PostgreSQL) nor Neo4j are able to handle well these immense graph explorations, although they have access to hundreds of gigabytes of memory. In particular, Neo4j spills to disk, hence the extreme performance difference compared to aDFS. Clearly, for graphs and queries at this scale, a fast graph-optimized solution such as aDFS, which easily handles these queries, is required. With the largest query (Q2a on Twitter) aDFS performs a 9.3T COUNT in 1,286 seconds, resulting in 7.3B matches per second, while consuming less than 10GB per-machine memory for intermediate results.

4.5 aDFS vs. Graph Mining, Dataflow Joins

Experiment. We compare aDFS to (i) three graph-mining systems,² namely *G-Miner* [27], *Fractal* [32], and *Peregrine* [42], as well as a dataflow join system, *BiGJoin* [19]. We use workloads from the G-Miner paper [27]: TC, i.e., Triangle Counting, and counting instances of a more complex pattern referred to as the P-pattern, with the four graphs that are used to evaluate these operations in the paper. All systems are distributed apart from Peregrine. For BiGJoin, we only perform the evaluation on TC as it does not support filters, and tune the batch size for performance (10^8). For aDFS, we express both triangles and the P-pattern as graph queries.

Results. Figure 13 includes the performance of the four systems. Triangle counting (TC) highlights the difference between matching and not matching automorphisms: For the three graph-mining systems, the search for “unique” triangles is baked in the pattern-matching algorithm, whereas in aDFS, we implement isomorphism with automorphism elimination using dynamic filtering (i.e., $(a) \rightarrow (b) \rightarrow (c) \rightarrow (a)$ WHERE $ID(a) < ID(b)$ AND $ID(b) < ID(c)$). This results in expensive filtering and heavier cross-machine communication than with the other systems. Still, aDFS is faster than G-Miner and Fractal for all graphs by up to 14× for G-Miner and by up to several orders of magnitude for Fractal. Peregrine outperforms all other graph-mining systems including aDFS on three out of the four graphs, as it is able to intersect adjacency lists to quickly find common neighbors, an optimization that

²We requested the artifact of Automine [54] for evaluation, but the authors were not able to provide us with it.

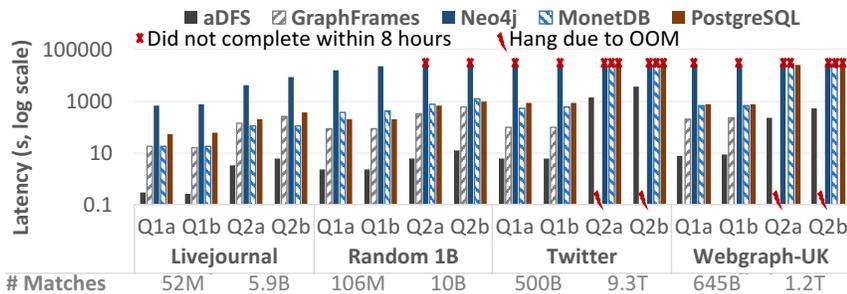


Figure 12: aDFS vs. other graph and relational systems on simple-pattern queries.

performs particularly well for triangles and which can be implemented in a straightforward manner on a single machine, where the whole graph is accessible. There is no clear winner between aDFS and BiGJoin on TC, with each system outperforming the other on two graphs. By intersecting local edges, BiGJoin’s approach allows for reduced communication and better performance on the two graphs with the highest average degrees (Orkut and Friendster).

The P-pattern does not require automorphism checks, as its vertices are differentiated by labels. We express it as:

```
(c:c) -> (b1:b) -> (:a) -> (c) -> (b2:b) -> (:d)
WHERE b1 <> b2
```

in PGQL. When matching the P-Pattern, aDFS significantly outperforms all other systems for all but one datapoint (G-Miner on BTC); it is on average 12 and 366× faster than Peregrine and Fractal, respectively, and 8× faster than G-Miner on three graphs. G-Miner achieves the best performance on BTC mainly because it replicates the target vertex label with each edge, which increases locality and reduces communication traffic. Such an optimization is not practical in a real-world system in which vertices can have many labels and properties of various types: Replicating these for each edge can have unacceptable memory overhead.

Overall, although aDFS is designed for different workloads, i.e., expressive graph queries, it is still very competitive with state-of-the-art graph-mining systems and a dataflow join system on triangle counting and/or a mining-oriented workload.

4.6 aDFS Scalability

Experiment and results. We use the LDBC workload to illustrate the scalability of aDFS as we vary the number of machines. Figure 14 includes the speedups, normalized to the latency of a single machine. Overall, aDFS exhibits

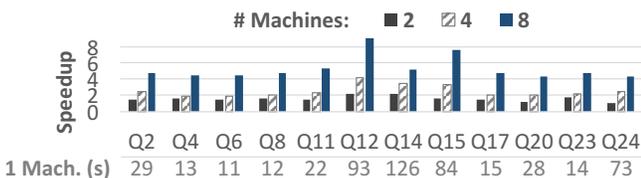


Figure 14: Scalability of aDFS vs. using a single machine.

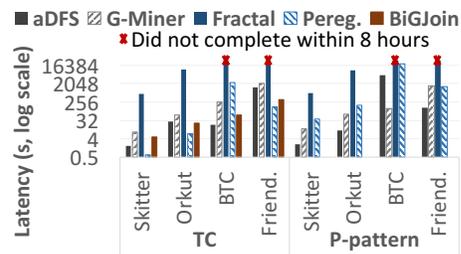


Figure 13: aDFS vs. graph-mining and dataflow join systems on triangles, patterns.

very good scalability: The average speedup is 1.6× from one to two machines, 2.5× from one to four machines, and 5.4× from one to eight machines. These numbers include various distributed coordination and query compilation overheads, as well as additional fixed costs. The core runtime of aDFS actually has even better scalability: looking at pure pattern-matching execution time, without coordination overheads, GROUP BY, and ORDER BY, the speedup improves to 1.7×, 2.6×, and 6× from one to two, four, and eight machines respectively (not shown). aDFS is designed to scale: More machines translate to more compute resources, more buffers for intermediate results, and often more BFS exploration and higher network utilization, as the percentage of remote edges increases with the number of machines.

5 Related Work

Database Management Systems (DBMSs). DBMSs offer graph support via a multi-model premise, but focus on SQL-like rather than pattern-matching querying [7, 12, 13, 52]. Kalinsky et al. [43] acknowledge that using DBMS joins for graph pattern matching is suboptimal, and propose hardware support to alleviate the issue. In contrast to DBMSs, aDFS is an efficient in-memory distributed graph-querying system that considers graph storage and queries as first-class citizens and focuses on analytical rather than transactional workloads.

Graph Algorithms. There is a plethora of related work for executing graph algorithms (such as PageRank [57]). Single-machine solutions focus on various topics such as proposing DSLs [38] or programming models [55] for graph algorithms, performance optimizations [65, 67], leveraging hardware features such as NUMA [81] and GPUs [56, 84], or supporting out-of-core computing [83]. Distributed solutions focus on topics such as asynchronous processing and performance [35, 50], efficient partitioning [78, 82, 85], leveraging hardware features such as RDMA [75], support for secondary storage [61], distributing sequential algorithms [33], approximate computing [70], alternative programming paradigms [76], or fault tolerance [30, 71]. aDFS focuses on graph queries rather than algorithms, but it shares features with some of these distributed solutions, such as the use of asynchronous processing or (random) graph partitioning.

Graph Querying. A number of single-node graph-querying systems were proposed by academia: Sun et al. [66] and Lin et al. [49] build relational and transactional systems, Graphflow [44] is an active graph database that supports evaluating one-time and continuous subgraph queries, TurboFlux [46] optimizes fast continuous subgraph matching over a fast graph update stream, and CECI [22] uses multiple embedding clusters and intersections of neighborhood lists to optimize subgraph matching (CECI can be distributed through graph replication/sharing, not graph distribution as with aDFS, due to the challenges mentioned in Section 2.4). In earlier work, we prototyped simple distributing DFS exploration [60].

There are numerous industrial graph-querying solutions. Neo4j [10] is single-machine and supports Cypher [11] queries. Amazon Neptune [1] is built for the Amazon cloud. Facebook Dragon [3] builds indices on updates for accessing data. Microsoft Graph Engine [8] is an in-memory data processing system based on Trinity [63], and TigerGraph [18] distributes GSQL [4] queries based on the source vertex data for a given query hop. Furthermore, there are also open-source distributed solutions. JanusGraph [6] uses distributed graph storage but does not distribute computation. GraphFrames [31] implements graph pattern matching with Spark using joins of dataframes. Wukong [64, 73] is a distributed graph-based RDF store that leverages hardware features, such as RDMA and GPUs, which we do not focus on. To the best of our knowledge, aDFS is the first truly distributed graph-querying system that works on fully-partitioned graphs and strictly bounds memory while maintaining great performance.

Graph-Mining systems. Graph-mining focuses on extracting structural properties and computing complex aggregate statistics [34, 74] of a graph by exploring its subgraph structures. Examples include triangle counting, maximal clique finding, community detection, and graph matching [27, 54, 59]. Graph-querying systems typically employ a vertex/edge-centric processing approach: A state is maintained per vertex and communicated to its neighbors [54, 69]. Graph-mining systems typically follow a subgraph-centric (often undirected and schema-less) processing approach: They attach information to a large amount of intermediate results composed of subgraphs [54] rather than specific vertices. Additionally, graph-mining systems typically leverage automorphism elimination [29, 32, 42], while while graph-querying engines generate homomorphisms to answer user graph queries.

Recent single-machine systems include RStream [72], AutoMine [54], and Peregrine [42]. Distributed systems include Arabesque [69], NScale [59], G-thinker [77], BiGJoin [19], G-Miner [27], ASAP [41], and Fractal [32]. aDFS shares features with some of these systems. For example, forms of asynchronous computations are used in G-Miner [27] (with a “task-pipeline” to hide communication overheads) and BiGJoin [19] (with data-parallel dataflow computations that pick up dynamically joined columns with the least matches). Techniques to reduce memory consumption are

used by G-Thinker [77] (buffering excess subgraph-tasks in a disk-based priority queue), BiGJoin [19] (primarily using batching to limit memory consumption but not for intermediate results as with aDFS) and Fractal [32]. Fractal combines a DFS strategy with a “from-scratch processing” paradigm which leads to re-computation overheads (absent in aDFS), as well as imbalances across workers that are mitigated by work stealing: workers break the DFS strategy to steal enumerations, which can be at any level of the matched graph pattern, from other workers. aDFS uses asynchronous DFS-based graph traversals together with flow control to strictly bound memory consumption, and can switch to BFS, in the same graph pattern-matching level, to generate more local work and to buffer remote edges (see Section 3). Our in-depth evaluation shows that the performance of aDFS for graph pattern-matching is competitive with that of state-of-the-art graph-mining systems.

BFS/DFS. The BFS/DFS tradeoff has been explored in the context of single-machine parallel task-scheduling runtimes. Typically, DFS is used to schedule a task graph in order to curtail memory [28], and BFS is used opportunistically (often called “work stealing”) to maximize parallelism [23, 24]. aDFS leverages these insights in the context of distributed graph query processing.

6 Concluding Remarks

Conclusions. We have introduced aDFS: A system that uses an efficient, almost-DFS approach to execute pattern-matching queries on distributed graphs. aDFS is able to execute virtually any query on any in-memory graph using at most a fixed, configurable amount of memory. aDFS is also very fast and scalable. We compared aDFS to eight state-of-the-art systems with diverse characteristics—graph or relational/join-based, distributed or single machine, in-memory or disk-based—and showed that aDFS is up to orders of magnitude faster than them.

Limitations and future work. aDFS uses simple algorithms for query optimization and graph partitioning, as this paper focused on runtime support for distributed graph querying. In the future, we will improve query planning and optimization, together with graph partitioning and caching. We will also consider query optimization opportunities to enable pruning of the traversal space when the underlying data has a relational-style schema, as described in Section 4.2.

Acknowledgments. Tomáš Faltín was partially supported by the Charles University, project GA UK No. 396721. We would like to thank our anonymous reviewers, as well as our shepherd, Keval Vora, for their feedback.

References

- [1] Amazon Neptune – Fast, reliable graph database built for the cloud. <https://aws.amazon.com/neptune/>.
- [2] Apache Spark – Unified analytics engine for big data. <https://spark.apache.org>.

- [3] Facebook Dragon – A distributed graph query engine. <https://code.fb.com/data-infrastructure/dragon-a-distributed-graph-query-engine/>.
- [4] GQL Standard – Graph Query Language. <https://www.gqlstandards.org>.
- [5] Gremlin – A graph traversal language. <https://github.com/tinkerpop/gremlin/wiki>.
- [6] JanusGraph – Distributed, open source, massively scalable graph database. <https://janusgraph.org>.
- [7] Microsoft Azure Cosmos DB – Fast NoSQL database with open APIs for any scale. <https://azure.microsoft.com/en-gb/services/cosmos-db/>.
- [8] Microsoft Graph Engine – Serving big graphs in real-time. <https://www.graphengine.io>.
- [9] MonetDB – An open-source database system. <https://www.monetdb.org>.
- [10] Neo4j – Graph database platform. <https://neo4j.com>.
- [11] Neo4j Cypher Query Language – Developer guides. <https://neo4j.com/developer/cypher/>.
- [12] OQGRAPH – The Open Query GRAPH engine for MariaDB. <https://openquery.com.au/products/graph-engine>.
- [13] OrientDB Community Edition. <https://orientdb.org>.
- [14] PGQL 1.1 Specification – Property Graph Query Language. <https://pgql-lang.org/spec/1.1/>.
- [15] PostgreSQL – The world’s most advanced open source database. <https://www.postgresql.org>.
- [16] SNAP: Network Datasets – Google web graph. <https://snap.stanford.edu/data/web-Google.html>.
- [17] SPARQL Query Language for RDF – SPARQL Protocol and RDF Query Language. <https://www.w3.org/TR/rdf-sparql-query/>.
- [18] TigerGraph Distributed Query Mode – Documentation. <https://docs.tigergraph.com/dev/gsql-ref/querying/distributed-query-mode>.
- [19] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed Evaluation of Subgraph Queries Using Worst-Case Optimal Low-Memory Dataflows. *PVLDB*, 2018.
- [20] Lars Backstrom, Daniel P. Huttenlocher, Jon M. Kleinberg, and Xiangyang Lan. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *SIGKDD*, 2006.
- [21] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5), 2001.
- [22] Bibek Bhattacharai, Hang Liu, and H. Howie Huang. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *SIGMOD*, 2019.
- [23] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably Efficient Scheduling for Languages with Fine-Grained Parallelism. *J. ACM*, pages 281–321, 1999.
- [24] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPOPP*, 1995.
- [25] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A Large Time-Aware Web Graph. *SIGIR Forum*, 42(2), 2008.
- [26] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking The Memory Wall in MonetDB. *Commun. ACM*, 51(12), 2008.
- [27] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-Miner: An Efficient Task-Oriented Graph Mining System. In *EuroSys*, 2018.
- [28] Shimin Chen, Todd C. Mowry, Chris Wilkerson, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, and Nikos Hardavellas. Scheduling Threads for Constructive Cache Sharing on CMPs. In *SPAA*, 2007.
- [29] Soumyava Das and Sharma Chakravarthy. Duplicate Reduction in Graph Mining: Approaches, Analysis, and Evaluation. *IEEE KDD*, 30(8), 2018.
- [30] Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. Phoenix: A Substrate for Resilient Distributed Graph Analytics. In *ASPLOS*, 2019.
- [31] Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. GraphFrames: An Integrated API for Mixing Graph and Relational Queries. In *GRADES*, 2016.
- [32] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A General-Purpose Graph Pattern Mining System. In *SIGMOD*, 2019.

- [33] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. Parallelizing Sequential Graph Computations. In *SIGMOD*, 2017.
- [34] Brian Gallagher. Matching Structure and Semantics: A Survey on Graph-Based Pattern Matching. In *AAAI Fall Symposium*, 2006.
- [35] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, 2012.
- [36] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph Processing in A Distributed Dataflow Framework. In *OSDI*, 2014.
- [37] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. TriAD: A Distributed Shared-Nothing RDF Engine Based on Asynchronous Message Passing. In *SIGMOD*, 2014.
- [38] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *ASPLOS*, 2012.
- [39] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Lugt, Merijn Verstraaten, and Hassan Chafi. PGX.D: A Fast Distributed Graph Processing Engine. In *SC*, 2015.
- [40] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11), 2011.
- [41] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. ASAP: Fast, Approximate Graph Pattern Mining at Scale. In *OSDI*, 2018.
- [42] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A Pattern-Aware Graph Mining System. In *EuroSys*, 2020.
- [43] Oren Kalinsky, Benny Kimelfeld, and Yoav Etsion. The TrieJax Architecture: Accelerating Graph Operations Through Relational Joins. In *ASPLOS*, 2020.
- [44] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An Active Graph Database. In *SIGMOD*, 2017.
- [45] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. Taming Subgraph Isomorphism for RDF Query Processing. *Proc. VLDB Endow.*, 8(11), 2015.
- [46] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *SIGMOD*, 2018.
- [47] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. What Is Twitter, A Social Network Or A News Media? In *WWW*, 2010.
- [48] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI*, 2012.
- [49] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. Fast In-Memory SQL Analytics on Typed Graphs. *PVLDB*, 10(3), 2016.
- [50] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J.M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 5(8), 2012.
- [51] Andrew Lumsdaine, Douglas P. Gregor, Bruce Hendrickson, and Jonathan W. Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(1), 2007.
- [52] Hongbin Ma, Bin Shao, Yanghua Xiao, Liang Jeff Chen, and Haixun Wang. G-SQL: Fast Query Processing via Graph Exploration. *PVLDB*, 9(12), 2016.
- [53] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *SIGMOD*, 2010.
- [54] Daniel Mawhirter and Bo Wu. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. In *SOSP*, 2019.
- [55] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *SOSP*, 2013.
- [56] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *ASPLOS*, 2018.
- [57] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford InfoLab, 1999.
- [58] Vijayan Prabhakaran, Ming Wu, Xuettian Weng, Frank McSherry, Lidong Zhou, and Maya Haradasan. Managing Large Graphs on Multi-Cores with Graph Awareness. In *USENIX ATC*, 2012.

- [59] Abdul Quamar, Amol Deshpande, and Jimmy Lin. NScale: Neighborhood-Centric Large-Scale Graph Analytics in the Cloud. *The VLDB Journal*, 25(2), 2016.
- [60] Nicholas P. Roth, Vasileios Trigonakis, Sungpack Hong, Hassan Chafi, Anthony Potter, Boris Motik, and Ian Horrocks. PGX.D/Async: A Scalable Distributed Graph Pattern Matching Engine. In *GRADES Workshop*, 2017.
- [61] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-Out Graph Processing From Secondary Storage. In *SOSP*, 2015.
- [62] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-Centric Graph Processing Using Streaming Partitions. In *SOSP*, 2013.
- [63] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *SIGMOD*, 2013.
- [64] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration. In *OSDI*, 2016.
- [65] Julian Shun and Guy E. Blelloch. Ligr: A Lightweight Graph Processing Framework for Shared Memory. In *PPoPP*, 2013.
- [66] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guo Tong Xie. SQL-Graph: An Efficient Relational-Based Property Graph Store. In *SIGMOD*, 2015.
- [67] Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Subramanya Dullloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. GraphMat: High Performance Graph Analytics Made Productive. *PVLDB*, 8(11), 2015.
- [68] Gábor Szárnyas, Arnau Prat-Pérez, Alex Averbuch, József Marton, Marcus Paradies, Moritz Kaufmann, Orri Erling, Peter A. Boncz, Vlad Haprian, and János Benjamin Antal. An Early Look at The LDBC Social Network Benchmark's Business Intelligence Workload. In *GRADES Workshop*, 2018.
- [69] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A System for Distributed Graph Mining. In *SOSP*, 2015.
- [70] Keval Vora, Rajiv Gupta, and Guoqing Xu. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *ASPLOS*, 2017.
- [71] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. CoRAL: Confined Recovery in Distributed Asynchronous Graph Processing. In *ASPLOS*, 2017.
- [72] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine. In *OSDI*, 2018.
- [73] Siyuan Wang, Chang Lou, Rong Chen, and Haibo Chen. Fast and Concurrent RDF Queries Using RDMA-Assisted GPU Graph Exploration. In *USENIX ATC*, 2018.
- [74] Takashi Washio and Hiroshi Motoda. State of the Art of Graph-Based Data Mining. *ACM SIGKDD Explorations Newsletter*, 5(1), July 2003.
- [75] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. GraM: Scaling Graph Computation to the Trillions. In *SoCC*, 2015.
- [76] Chengshuo Xu, Keval Vora, and Rajiv Gupta. PnP: Pruning and Prediction for Point-To-Point Iterative Graph Analytics. In *ASPLOS*, 2019.
- [77] Da Yan, Hongzhi Chen, James Cheng, M. Tamer Özsu, Qizhen Zhang, and John C. S. Lui. G-thinker: Big Graph Mining Made Easier and Faster. *CoRR*, abs/1709.03110, 2017.
- [78] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs. *PVLDB*, 7(14), 2014.
- [79] 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.
- [80] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A Distributed Graph Engine for Web Scale RDF Data. *PVLDB*, 6(4), 2013.
- [81] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-Aware Graph-Structured Analytics. In *PPoPP*, 2015.
- [82] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the Hidden Dimension in Graph Processing. In *OSDI*, 2016.
- [83] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System. In *ASPLOS*, 2018.

- [84] Yu Zhang, Xiaofei Liao, Hai Jin, Bingsheng He, Haikun Liu, and Lin Gu. DiGraph: An Efficient Path-Based Iterative Directed Graph Processing System on Multiple GPUs. In *ASPLOS*, 2019.
- [85] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *OSDI*, 2016.
- [86] Lei Zou, M. Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, and Dongyan Zhao. gStore: A Graph-Based SPARQL Query Engine. *The VLDB Journal*, 23(4), 2014.

GLIST: Towards In-Storage Graph Learning

Cangyuan Li ^{1,2}, Ying Wang ^{1,2}, Cheng Liu ^{1,2}, Shengwen Liang ^{1,2}, Huawei Li ^{1,2,3}, Xiaowei Li ^{1,2}
SKLCA, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China ¹
University of Chinese Academy of Sciences, Beijing, China ²
Peng Cheng Laboratory, Shenzhen, China ³

Abstract

Graph learning is an emerging technique widely used in diverse applications such as recommender system and medicine design. Real-world graph learning applications typically operate on large attributed graphs with rich information, which do not fit in the memory. Consequently, the graph learning requests have to go across the deep I/O stack and move massive data from storage to host memory, which incurs considerable latency and power consumption. To address this problem, we developed GLIST, an efficient in-storage graph learning system, to process graph learning requests inside SSDs. It has a customized graph learning accelerator implemented in the storage and enables the storage to directly respond to the graph learning requests. Thus, GLIST greatly reduces the data movement overhead in contrast to conventional GPGPU based systems. In addition, GLIST offers a set of high-level graph learning APIs and allows developers to deploy their graph learning service conveniently. Experimental results on an FPGA-based prototype show that GLIST achieves $13.2\times$ and $10.1\times$ average speedup and reduces the power consumption by up to 98.7% and 98.0% respectively on a series of graph learning tasks when compared to CPU and GPU based solutions.

1 Introduction

Graph is a fundamental data structure widely seen in modern computer systems and applications. Real-world social networks, molecular graph structures, biological protein networks, social networks, and data from many other fields can be modeled as graphs, particularly the attributed graphs (AGs), which carry richer property information than well-studied plain graphs [19, 40, 42]. Attributed graphs occupy a growing proportion of storage space in the datacenters of service providers such as Facebook, Amazon and Alibaba, and the trend will continue especially with the popularity of graph database and graph analytics platforms for citation networks and recommender systems [4, 7, 43, 58]. Taobao, one of the

largest online consumer-to-consumer (C2C) platforms, for example, manages attributed graphs that consist of one billion users and two billion items [43]. Therefore, as the machine learning technology advances, the question of how to make prediction, discover new patterns, and mine useful information from such rich attributed graphs, which is known as the area of Graph Learning (GL), is gradually becoming important in private and public cloud datacenters where the massive graph data can be ingested to learn the basic classification, clustering, visualization and prediction functionality [5, 29, 35, 43, 52–54].

Conventionally, common graph learning tasks require numerous CPU or GPU nodes to deal with large-scale graph learning problem and the related user queries, which directly translates to sheer growth of power and cost overhead. For instance, a typical GL-based recommender system in Alibaba [43] employs hundreds of GPUs in service to mine billion-scale attributed graph data associated to numerous customers and shopping items. To investigate more cost-effective GL systems, in this work we first characterize the real-world GL applications by building a conventional single-node GPU+SSD based graph learning system. In this system, several critical tasks found in realistic datacenter infrastructures are implemented and simulated (See Section 3). We discovered that there are several important impactful performance in these mainstream graph learning tasks. (1) For typical graph learning systems that respond to graph analysis requests as shown in Table 2, the storage-and-compute decoupled systems are bottlenecked by I/O operations, and they are not energy efficient in dealing with the GL requests due to the costly data movement from the storage to CPUs/GPUs. (2) Large-scale graph learning tasks exhibit poor data locality, which can hardly be exploited in the limited on-chip or even off-chip memory due to the large footprint of attributed graphs such as social networks or recommender systems. (3) We found that, although graph learning tasks are much more complicated than plain-graph processing, they are generally solvable by emerging graph neural networks (GNN), which means a compact specialized GL accelerator is a viable alter-

native to GPUs and CPUs in storage-centric GL systems.

To replace the power-hungry CPU/GPU based solutions and eliminate the unnecessary power consumed by graph data movement, we propose a near data computing system to realize efficient Graph Learning In-Storage (GLIST). As depicted in Figure 1(b), GLIST is a combination of in-SSD computing and customized graph learning accelerator architecture, and it enables the storage device to directly respond to attributed graph analysis requests and queries, making the data warehouse machines more energy efficient.

However, fitting large-graph learning tasks into compact storage devices remains very challenging and worth investigating. First, large graphs generally have too large footprint to fit in the DRAM memory or the caching memory of storage devices [20], thus processing a large attributed graph on request tends to have poor locality, which must be well exploited in the design of GLIST. In the graph learning process, how to efficiently and directly fetch graph learning model parameters and the graph itself from the flash devices, how to preserve locality in the working-set of GL, and how to exploit the abundant channel-level flash bandwidth in SSDs is also very important.

Second, fitting large-scale graph learning workloads into storage SSDs is challenging due to the limitation of power and computing resource inside the SSDs that generally have embedded CPU or MCU for flash device management, because deep learning technology based graph analysis workloads are bandwidth and computational intensive at the same time. This calls for a more efficient architecture to practice in-storage graph learning with SSDs.

Third, though analyzing a single graph request does not exhibit good memory locality, it is found that the inter-request locality does exist as the working-sets of temporally correlated requests overlap with one another to some degree as will be discussed in Section 3. Thus, to achieve the best efficiency of the in-storage computing, fully exploiting the concurrency and the inter-request locality in the graph analysis requests is also important. As a consequence, more consideration should be given to the working-set caching and the request scheduling strategy in the GLIST controller to reuse GL model and graph data in storage.

In all, we make the following contributions in this paper:

- We profiled real-world GL workloads in different categories and obtained two main observations for optimizing the architecture of GNN systems in terms of data locality, especially for systems with block-based storage devices.
- Based on our observations, we proposed the GLIST architecture to enable high-throughput graph learning services. We handle concurrent requests issued to the power-limited graph learning storage with specialized caching system and locality-centric request scheduling policy to exploit the data locality in and between the attributed

graph analysis requests. The graph analysis requests are processed inside SSDs with a unified hardware accelerator to handle various graph learning tasks instead of going across a deep I/O stack. To the best of our knowledge, GLIST is the first in-storage acceleration system for graph learning workloads such as recommender systems and automated customer service.

- We build a GLIST prototype on the Cosmos Plus OpenSSD platform [1]. Experimental results show that the GLIST caching and scheduling policy can improve the performance by up to 13.2× and 10.1× compared to CPU+SSD, GPU+SSD based system, respectively.
- GLIST provides a software abstraction with a set of programming APIs that enable developers to create and deploy their graph learning models and analysis service into the in-storage graph learning system.

2 Background

2.1 Graph Learning Tutorial

Graph neural network applications can be modeled as an encoding-decoding method [15, 58]. The encoding function encodes the vertices in a graph into latent representation (also called embedding) that summarizes both the location and neighboring information. The decoding function decodes the embedding to the original vertex information, which is directly related to graph learning tasks, such as labeling a vertex in classification task.

Table 1: GNN Notations.

Notations	Description	Notations	Description
\mathbf{G}	attributed graph $G(V,E)$	$Nb(v)$	vertex v 's neighbor set
\mathbf{h}_v	the embedding vector of vertex v	$N(v)$	subset of vertex v 's neighbor set
$e(i,j)$	the edge between v_i and v_j	R	analysis result

Typically, the encoder function is composed of three types of functions including **Sample**, **Aggregate**, and **Combine**. **Sample** controls the scope of the information to be processed in a graph. As formulated in Eq. 1, it samples a subset of the neighbor vertices and constructs a new sub-graph for embedding [6, 14, 53]. The notations used in the formulation is summarized in Table 1. **Sample** can also be omitted according to GCN [21] and GIN [48]. In this case, all the neighbor vertices are used for embedding calculation.

$$\mathbf{S}_v = \mathbf{Sample}^k(Nb(v)) \quad (1)$$

Aggregate aggregates the features of all the incoming vertices to update the feature of current vertex v .

$$\mathbf{h}_v^k = \mathbf{Aggregate}(\{\mathbf{h}_u^{(k-1)}\}_{u \in N_v}) \quad (2)$$

where $\mathbf{h}_v^{k'}$ is the feature of vertex v aggregated from features of neighbor vertices $\mathbf{h}_u^{(k-1)}$ at the $(k-1)$ th layer.

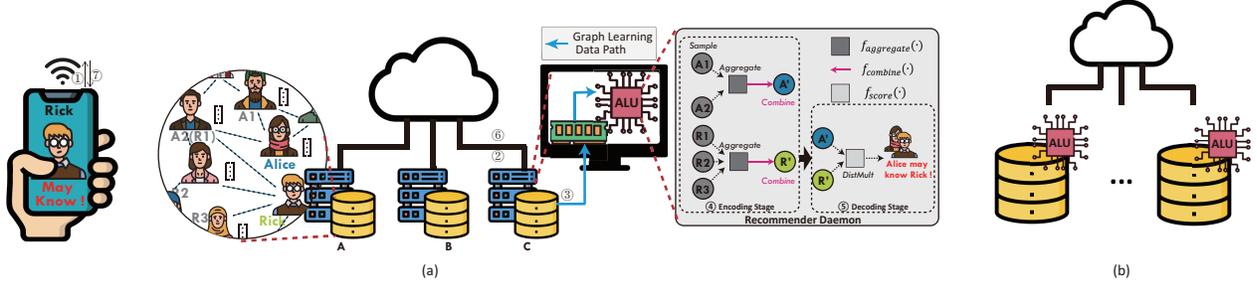


Figure 1: **A processing example of typical GL-based social network recommendation service [14, 21, 46]** When Alice tries to extend her connection via a social network App, a request is generated by the App and sent to the data center①. The request is then converted to multiple graph analysis operations to predict users who Alice may be interested in. One of the operations is assigned to Server C② to predict the potential connection between Alice and Rick. With conventional storage, the relevant sub graph of the huge social network must be loaded from the external storage to main memory and will be processed with an encoder function on host. The processing is to generate embedding vectors that can represent the two users’ social network characteristics③. Then a DNN-based predictor is invoked as a decoder to determine whether the two users may agree to connect④. Finally, the recommendation is obtained based on all the prediction results and sent to the user App⑤. (b) GLIST can simplify the graph learning processing. There is no data movement between the storage and the host system.

In order to obtain the updated feature of vertex v at layer k i.e. \mathbf{h}_v^k , **Combine**, which is essentially an MLP operation, is applied.

$$\mathbf{h}_v^k = \mathbf{Combine}(\mathbf{h}_v^k) \quad (3)$$

With multiple iterative processing, the obtained embedding vectors are fed to the decoder function to perform graph analysis tasks. The graph learning tasks can be categorized into three types [47]: **Node-level analysis**, **Edge-level analysis**, **Graph-level analysis**. Meanwhile, the decoder function varies from specific graph learning tasks, and they will be detailed as follows.

Node-level analysis aims to classify nodes without labels in graphs. It can also be applied for the classical community classification task in online social network analysis [3, 26], which essentially classifies nodes into several communities. The decoder function of **Node-level analysis** can be formulated as Eq. 4 [21].

$$\mathbf{R}_v = \mathbf{Decoder}(\mathbf{h}_v) \quad (4)$$

Edge-level analysis focuses on the prediction of missing edges or edges’ attributes. A typical use is to predict the potential connections between users and items in recommender systems, revealing a user’s interest in an item. The decoder function of **Edge-level analysis** can be formulated as Eq. 5 [38].

$$\mathbf{R}_{e(i,j)} = \mathbf{Decoder}(\mathbf{h}_i, \mathbf{h}_j) \quad (5)$$

Graph-level analysis operates on the entire graph as formulated below [48] and mainly targets for graph classification.

$$\mathbf{R}_G = \mathbf{Decoder}(\{\mathbf{h}_u\}_{u \in G}) \quad (6)$$

GNN can be used for many applications as summarized in Table 2. We take a GL-based social network recommender system shown in Figure 1 (a) as an example to illustrate the

use of GL. When the server receives an edge analysis request to predict the potential relationship between two users②, the algorithm will load the relevant sub-graphs of the social network in external storage with a **Sample** function to host memory, and then generates embedding vectors for the analyzed users③ with an **Aggregate** function and a **Combine** function. This will lead to high processing latency mainly caused by (1) random data access to storage and (2) massive data transfer across the bandwidth-limited PCIe bus and deep OS software stack. Then, the two generated embedding vectors are used by a DNN-based predictor to determine the existence of social connection between the two users④. Finally, the prediction result, which is usually a scalar, is sent back to the host machine and the user App eventually to make a recommendation. The proposed GLIST system, as shown in Figure 1 (b), however, performs the graph learning tasks only in SSDs to mitigate the drawbacks mentioned above.

2.2 In-storage Graph Processing

By enabling computation in storage that can avoid massive data movement between storage devices and host memory, in-storage computing (ISC) has become a promising computing paradigm for big data processing [8, 11, 17, 18, 28, 32, 37]. Graph processing on large-scale graphs is considered to be I/O intensive and requires frequent accesses to the graph in storage, so it fits well to the ISC paradigm. A number of prior works have intensively investigated the use of ISC for graph processing and demonstrated competitive performance and energy efficiency [18, 22, 23, 25, 31, 34, 41, 56]. GraphSSD [34] proposed a semantic-aware translation layer for efficient data access in graph processing. GraphOne [23] proposed an efficient dynamic graph store to facilitate both runtime graph update and processing. It supports various graph processing operations from distinct perspectives. G-Store [22] and MO-

Table 2: Graph learning tasks, algorithms, and datasets.

Analysis level	Model	Graph	#Vertices/#Graphs	#Edges(per graph)	Application
Node-Level	GCN [21]	ogbn-products (OP) [16]	2,449,029	61,859,140	Product category prediction
	GS-Pool [14]	soc-LiveJournal1 (SL) [3, 26]	4,847,571	68,993,773	On-line community classification
		twitter (TW) [24]	61,578,417	1,468,365,182	User classification in social network
Edge-Level	GS-Pool [14] PinSage [53]	ogbn-papers100M [16]	111,059,956	1,615,685,872	Research papers classification
		ogbl-citation2 (OCi) [16]	2,927,963	30,561,187	Missing citations prediction
		ogbl-wikikg2 (OW) [16]	2,500,604	17,137,181	Knowledge graph completion
Graph-Level	GCN [21] GIN [48]	SOC-Friendster (SF) [51]	65,608,366	1,806,067,135	Missing relationships prediction in social network
		ogbg-molpcba (OM) [16]	437,929	28.1	Molecular property prediction
		ogbg-code(OCO) [16]	452,741	124.2	Code summarization
		ogbg-ppa (OP) [16]	158,100	2,266.1	Taxonomic prediction

SAIC [31] also achieved efficient in-storage graph processing with redundant data elimination methods and locality optimizations. Graphene [30] and FlashGraph [57] were proposed to address the I/O challenge in graph processing by managing frequently accessed data in DRAM.

However, due to the power constraint, the low-end processors in storage usually have limited computing capability to deal with complicated and demanding tasks. In this case, many powerful hardware accelerators are built in the context of ISC in recent years. GraFBoost [18] develops a specialized accelerator to coalesce the random accesses to the storage in large-scale graph processing and achieves server-class performance with small memory and low power footprint. ExtraV [25] utilizes a cache-coherent hardware accelerator to achieve both high performance and high flexibility for plain graph analysis.

While prior in-storage graph processing works mainly target to analyze plain graphs which only have simple scalar attributes, they cannot fulfill the processing requirements of the graph learning workloads that mostly operate on graphs with large vector attributes, because the graph learning tasks have distinct data access patterns and computation intensity. In addition, the primitive operations used in graph learning can also be unique. For example, the **Sample** function is not supported by any of the conventional plain graph processing abstractions [33, 36, 55]. Thus, we are motivated to investigate a novel ISC architecture for cutting-edge learning tasks on large and sparse graphs.

3 GL Workload Study for GLIST Design

3.1 Single Workload Characterization

Experimental setup In order to characterize and gain insight of various graph learning workloads, we conduct an in-depth study on a series of real-world representative GL applications on GPU [44]. The details of the applications and the datasets used for evaluation are illustrated in Table 2. The models and evaluation datasets are all stored in a 1 TB Samsung 970 PRO NVMe SSD. The computation device is an NVIDIA V100 GPU (Volta) equipped with 16 GB HBM2 memory.

Result analysis The latency of a graph learning task is broken down into three parts: GPU compute time(Computation)

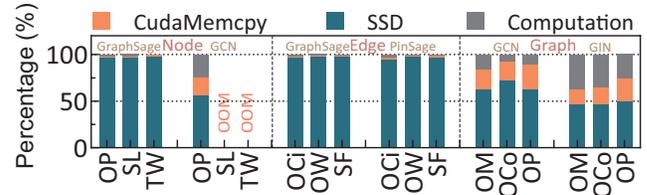


Figure 2: Performance breakdown of compute and I/O time for different graph learning workloads.

which implies the computation overhead, CudaMemcpy time(CudaMemcpy) which represents the time spent on the data movement between GPU and host, and the SSD to DRAM communication time (SSD). Figure 2 shows the profiling results, from which we can safely land two conclusions: First, the I/O bottleneck significantly penalizes the performance of graph analysis requests for most graph learning tasks. As we can see, all the graph learning tasks evaluated in this experiment spend more than half of their execution time on I/O operations, which means that GL workloads are limited by I/O bandwidth.

Second, it is hard for real-world large graphs to fit the memory. For example, twitter (TW) and soc-LiveJournal1 (SL) in the tested datasets cannot be fully loaded into 16 GB GPU memory. Thus, the algorithms have to load data from last-level storage on demand (Sampling based algorithms, i.e. GS-Pool and PinSage) or even cannot run (Non-Sampling based algorithms, i.e. GCN and GIN). Huge graphs not only increase the I/O access overhead, but also seriously restrict the throughput of general purpose processors, due to the memory capacity limitation. However, if the working set of graphs can be preprocessed where they originally stay and only the relevant sub-graph are moved, the data movement and processing overhead can be significantly reduced.

3.2 Locality in Graph Learning Workloads

To enable graph learning inside storage and service GL requests from users, we must exploit the locality in graph learning workloads to alleviate the SSD bandwidth limitation while preventing the long-latency flash accesses from penalizing the response performance. Two major exploitable locality observations that help fit the graph learning workloads into SSDs will be illustrated as follows:

1. *There exists working set reusability in between graph learning requests.*

Because SSD typically has much longer latency and coarser access granularity like pages and blocks, it is essential to take advantage of the limited DRAM or SRAM cache in it and exploit the locality in between the graph analysis requests for performance improvement. There are two potential types of locality in between the graph learning requests. The first type is Graph Data Locality (GDL) while the second is model parameter locality (MPL). For GDL, processing each vertex in the graph will involve a working set consists of its neighbors' property data. Graph analysis requests that happen to hit vertices in the proximate regions in the graph probably share a common working set. For MDL, many graph learning requests like node classification may utilize the same model parameters, so it will be beneficial to select and combine the graph learning requests with the same model parameters from all the batched requests.

2. *The layout of graph data in flash channels significantly impacts the locality of in-storage graph learning.*

Each single vertex/edge feature vector in attributed graphs is usually at the size of hundreds of bytes or few KB [19, 39, 50] and is smaller than a flash page size (i.e. 16 KB), the minimum operation granularity of flash devices. However, recent graph neural networks that respond to GL requests usually adopt **Sample** function which samples a subset of the target vertices' multi-hop neighbors [10, 14, 53]. This means that there may exist bandwidth under-utilization when the vertices located in the same flash page are not sampled simultaneously because the multi-hop structural correlation may not be captured by the **Sample** function.

For *Observation 1*, the request scheduling and caching strategy should be designed to fully exploit the temporal data locality that exists between requests. For *Observation 2*, the feature data layout in flash devices should be reorganized to improve the data reuse in a flash page.

4 GLIST Design

4.1 System Overview

To move the graph learning ability into storage devices, a state-of-the-art GL framework must support for system designers to develop and deploy the service of GL functions, e.g. GNN-based recommender systems and vertex classification, in storage devices. Inspired by the GL framework described in [58], we construct a multi-layered system architecture for the GLIST system, including user interface, run-time management, and specialized hardware as shown in Figure 3.

For the purpose of processing various graph data with GLIST, users can interact with it using the provided commands (see Table 3) via the GLIST Application Interface to define or invoke the specific GL functions in storage devices. Except the interface of defining and calling graph analysis

functions in storage, GLIST also implicitly performs locality-aware graph reorganization for the newly registered and updated graphs on the host machine, so that the GLIST system can improve storage operations and the response efficiency when processing the received user analysis requests.

Take the recommender system shown in Figure 1 as an example. At the offline stage, the social network graph which embeds the users' friendship information with connected edges is registered and stored in the flash devices of the GLIST system by *GraphRegister()*. The API also quantizes the vertex feature vector and chooses appropriate bit-width for edge data representation. The registered graph will further be used to make recommendation via GNN algorithms by predicting the existence of an edge. The GNN-based recommender model, e.g. PinSage from Pinterest [53] is trained and obtained by the application developers, and is then registered and kept in storage via *ModelRegister()*. It will be later invoked on requests.

After the model deployment, the users' clicks on the relevant App are converted to GL queries and sent to the data center machines. Particularly for the friend recommendation queries, essentially they belong to typical link predictions over the social network graph and will be handled by the daemon process running on the host of the GLIST system. On receiving the requests, the daemon process calls *GraphAnalysis()*. To exploit the data reuse between requests and ensure the request processing latency at the same time, the GL requests are batched in fixed time windows before being issued to the computing storage in GLIST ①. In the computing storage, a runtime environment is maintained to manage the incoming link prediction requests②. It translates each link prediction request to primitive analysis commands including a vertex embedding command that invokes the encoder function and a prediction command that executes the decoder function. The link prediction can be obtained after the execution of the corresponding primitive analysis commands.

In addition, the GLIST runtime also provides optimizations to exploit the data reuse within the batched requests and roughly includes two parts: (1) It reorders the primitive vertex analysis commands that generate flash accesses (i.e. vertex embedding requests) to explore the graph data reuse and fits the flash accesses to the flash channel-level parallelism. (2) It groups the reordered primitive vertex analysis commands into small batches to increase the bandwidth utilization of ways and channels, instead of sequentially handling each graph analysis command with limited footprint [14, 53]. After the commands are received and handled by the GLIST runtime, they are further decoded and sent as instructions to the GLIST processor that eventually executes and accelerates the graph learning functions. The instructions are served by the Sampler first, which fetches the feature vectors from the flash devices ⑦-⑧ or the Page Cache directly ⑨ from the on-board DRAM. Then, it constructs a larger sub-graph by merging multiple small sub-graphs obtained from the grouped

Table 3: GLIST APIs

Category	APIs
Graph Update	AddEdge, RemoveEdge, AddVertex, RemoveVertex, UpdateVertex
Graph Registration	GraphRegister, Graph Unregister
Model Registration	ModelRegister, ModelUnregister
Graph Analysis	GraphAnalysis, GetAnalysisResult

sampling functions [6, 14, 43]. The newly assembled graph is further loaded to the on-chip buffers of GLA. When all the required data are ready, the processing element array is instructed by the commands to execute the invoked vertex analysis model. Afterwards, when the feature vectors of the queried edge’s endpoints are ready, the primitive prediction function is scheduled onto the GLA to predict the link probability between target vertices. Finally, the GLA triggers the GLIST Runtime to collect the results and return the analysis results to the daemon process via *GetAnalysisResult()*.

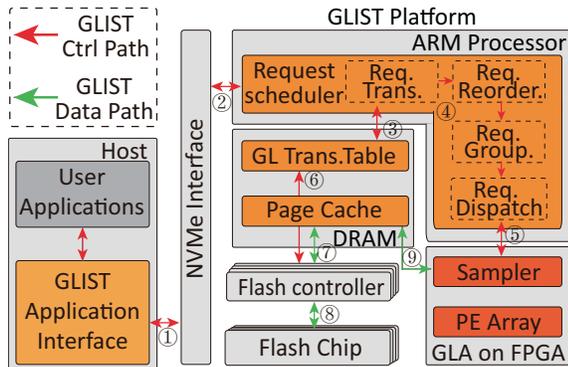


Figure 3: System architecture of GLIST.

4.2 The GLIST Runtime

The GLIST runtime is designed to decode, schedule, and issue the input requests to flash devices and the GLA. It manages the incoming requests as commands from the host machine, and also exploits the locality in between concurrent requests, and re-schedule the requests to maximize the available locality. To improve the flash bandwidth utility and exploit the inter-request locality, the GLIST runtime maintains two key structures, the Page Cache and the Graph Learning Translation Table, which enable the reuse of graph data and GNN models fetched from the flash devices in and between consecutive requests.

GLIST Page Cache is adopted to exploit temporal data locality between user requests. It caches the edges, vertex feature vectors and model data touched by the previously executed requests. Besides, the intermediate data such as the embedding vectors of vertices are also cached. We adopted the Least Recent Use strategy as replacement policy. The Page Cache works in the process of request response, and it is also used to hide the latency of operations correlated to the GNN function deployment stage, e.g. registering new function models.

Graph Learning Translation Layer, denoted as GL-TL, is provided to index reusable objects in SSDs including the graph property data, edge information, and analysis model parameters. GL-TL replaces the conventional LBA-to-PPN (logic block address to physical page number) paging used in commercial SSDs. GL-TL includes three translation tables, i.e. the Vertex Mapping Table, the Property Mapping Table, and the Model Mapping Table. The Vertex Mapping Table records the mapping between the vertex ID and the flash page which keeps its neighbors. Besides, it also records other meta-data of each vertex such as the number of adjacent vertices. Similarly, the Model Mapping Table and the Property Mapping Table keep the logical object index and physical block address. All the tables are kept in the DRAM when GLIST is activated.

4.2.1 In-storage Graph Learning Request Scheduling

Though the GLIST Page Cache and GL-TL enable the reuse of graph in between requests, how to group the requests into batch of concurrently executed commands impacts the efficacy of locality enhancement. When being requested, GNNs usually at first sample the large graph and operate on certain sub-graphs. Due to the random sampling strategies, analyzing a single vertex usually touches several flash pages to fetch the feature vectors of the sampled sub-graph, which has very unpredictable locality and sometimes causes a huge waste of flash bandwidth. However, if multiple analysis requests are concurrently processed and tactically reordered by the GLIST runtime, the flash bandwidth utilization will be improved. Nevertheless, due to the limited size, the DRAM in storage cannot accommodate the whole working set of a large request batch. Reordering and grouping the requests will help improve the cache reusability. In this way, multiple groups are served sequentially to reuse the shared data including the attribute information and intermediate data of vertices, because the groups of different requests may overlap with one another and share the intermediate or the input property data in the requests. Moreover, the requests in each group are fused and processed as a batch can better utilize both the flash bandwidth and the PE array of GLA.

The process of GL requests scheduling is shown in [Algorithm 1](#). To exploit the intermediate data reuse, GLIST leverages an encoding-decoding manner by splitting the GNN workflow into vertex embedding phase and prediction phase. In vertex embedding phase, the intermediate data can be reused by other analysis requests. For example, as shown in [Figure 1](#), the latent representation of each user obtained from this phase can be used to generate any recommendations related to that user. The operations in the prediction phase, however, are highly dependent on each specific user request, which hardly share intermediate data. Therefore, GLIST parses the requests (Line 3), so that only the primitive vertex embedding requests are re-scheduled. After re-scheduling, the primitive

Algorithm 1: Request Scheduling

```
Input: Graph  $G$ , Request  $R_i$ , Group Size  $S$   
Output: Scheduled Requests  $R_o$ , Embedding-Prediction Mapping Table  $EP\_MT$   
1  $req\_mapping\_table = \text{dict}()$   
2  $par\_ri = \text{usr\_req\_partition}(R_i)$   
3 for  $user\_req \leftarrow par\_ri$  do  
4   if  $user\_req.type == "Edge"$  then  
5      $primitive\_req\_mapping\_table =$   
6        $\text{extract\_edge}(primitive\_req\_mapping\_table, user\_req)$   
7   else if  $user\_req.type == "Graph"$  then  
8      $primitive\_req\_mapping\_table =$   
9        $\text{extract\_graph}(primitive\_req\_mapping\_table, user\_req)$   
10  else  
11     $primitive\_req\_mapping\_table =$   
12       $\text{extract\_vertex}(primitive\_req\_mapping\_table, user\_req)$   
13  $reordered\_primitive\_req, primitive\_req\_mapping\_table =$   
14    $\text{reorder\_primitive\_req}(task\_primitive\_req, primitive\_req\_mapping\_table)$   
15  $R_o, EP\_MT = \text{request\_grouping}(\text{Config}, reordered\_primitive\_req,$   
16    $primitive\_req\_mapping\_table)$   
17 return  $R_o, EP\_MT$ 
```

requests are reordered, and an embedding-prediction mapping table (EP_MT) will be used to record the mapping information between the re-ordered embedding phase and the intact prediction phase, so that GLIST can correctly execute the prediction phase.

Requests decomposition and reordering. The sub-graphs associated to the GL requests may overlap with each other to different extent, and contribute to different degree of locality. Thus, to maximize the temporal data locality in-between embedding requests that hit the same graph, the runtime scheduler reorders the primitive requests according to the affinity of their sampled regions in the graph. Though there are different categories of reusable data worth exploiting, the scheduler prioritizes the reuse of intermediate embedding data over that of input property data. For example, the requests on graph edge analysis, all begins with the embedding of the endpoints that can be reused while the latter prediction can be done independently. Thus, the Edge-level analysis request is decomposed by the scheduler into two primitive **Node-level Analysis** requests and one prediction request. In this way, the requests in embedding phase can be scheduled to maximize data locality and the prediction phase of different requests can reuse the embedding vectors of vertices in the DRAM cache. As shown in Algorithm 1, the batch of requests is initially separated into the primitive requests on vertices according to the requested graphs and the type of requests (Line 2). Then, the scheduler scans through the requests and reorders the primitive requests that may hit different sub-graphs (Line 4 — Line 9). Because each primitive request is correlated to a vertex as the analysis target, estimating the locality in-between requests is to measure the size of overlapping area of the corresponding vertex sub-graphs, which includes all the vertices that are n -hops away from the target vertex. Thereby, in scheduling, the $reorder_primitive_req()$ function is used to obtain the vertex whose sub-graph share more vertices with the previous scheduled requests than others. In practice, we

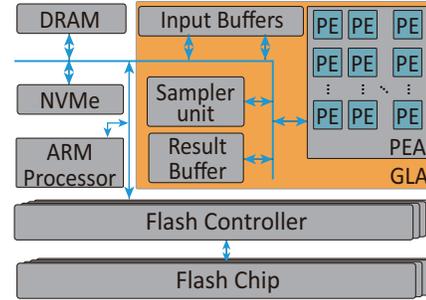


Figure 4: The GLA architecture and its integration to the hardware system.

implement the function by simply finding the vertex that has the minimum distance from the previously requested vertex.

Requests grouping. As mentioned above, the sub-graph obtained by the Sample function for each requested vertex usually causes random but small-footprint memory access, which tends to cause low flash bandwidth. In addition, for most of the GNN models, the bottom layers are witnessed to contribute less computation overhead than the top layers [6, 14, 43]. As a result, a single request hitting a vertex may not fully utilize the Processing Element (PE) resources of the GLA especially in the last layer of the GNN models. Therefore, GLIST batches all the reordered requests obtained from the request reordering stage into several groups as described in Line 11 of Algorithm 1. By fusing multiple requests of the same tasks into a batched task, both the utilization of flash bandwidth and the GLA are improved.

4.3 In-storage Graph Learning Accelerator

4.3.1 The Accelerator Architecture

Based on the design presented in [27], the Graph Learning Accelerator presented in Figure 4 is composed of a graph Sampler, on-chip buffers, and a Processing Element Array (PEA) to perform graph neural network inference.

The Sampler unit. For attributed graph analysis, the Sampler unit samples the vertices and edges from a large graph according to the predefined manner, before invoking GNN inference. It supports uniform distribution sampling or other predefined sampling functions [6, 14, 53]. In the sampling stage, the property data and their connection of the sampled vertices under request are loaded from flash devices to the DRAM of the GLIST embedded platform and further to the corresponding on-chip buffers. Besides, the Sampler in GLIST also supports non-sampling based GNN models [21, 49] by loading tiled graph sequentially according to the predefined tiling configuration.

According to the GNN framework introduced in Figure 2.1, the **Combine** and **Aggregate** functions should also be supported by the GLA. A Processing Element Array (PEA) is designed to address the matrix operations in **Combine** function. Each column of the PEA handles a single dimension

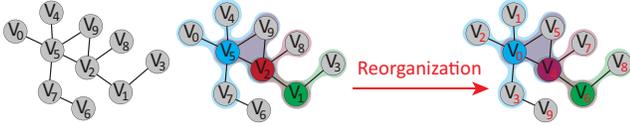


Figure 5: Graph reorganization.

of the input property for all the input vertices while the PEs in the same row are dedicated to one single vertex, so that the PEA structure is independent to the dimension of graph vertex properties.

To support the **Aggregate** function, we adopted a full mesh topology in our design by fully connecting columns of PEs in the array to achieve high-throughput message passing. Each PE in the same column broadcasts its data to all other columns and select data from other PEs’ output according to the control signal generated by the controller in PEA.

Algorithm 2: Graph Reorganization

Input: Graph G , Hop count h , Degree threshold Td , Center vertex number threshold Cn
Output: Reorganized Graph

```

1 important_vertices = Top  $Cn$  nodes whose in degree  $< Td$  of  $G$ 
2 for  $i \rightarrow \text{important\_vertex}$  do
3   | workingset[ $i$ ] = Sample from workingset[ $i$ ]
4 end
5 vertex_sequence = []
6  $i = 0$ 
7 while workingset is non-empty do
8   | Add vertex  $j$  that has the maximum intersection with vertex  $i$  in
9   | workingset[] to vertex_sequence
10  | Remove workingset[ $i$ ] from workingset
11  |  $i = j$ 
12 end
13 map_table = map()
14 for  $i \rightarrow \text{vertex\_sequence}$  do
15   | Assign new  $ID$  to vertex  $i$  and its  $h$  hop neighbors
16   | Record the mapping information in map_table
17 end
18 new_graph = Construct new graph with  $G$  and map_table
19 return new_graph

```

4.3.2 Graph Reorganization

When analyzing a vertex in a Sampling based graph learning workflow, its closer neighbors are more likely to be accessed, which shows the existence of spatial locality in GL workloads. However, the property of the vertices usually takes hundreds of bytes or few KB, which is much smaller than a flash page size and may cause flash bandwidth under-utilization. Therefore, we designed a heuristic algorithm to re-index the vertex IDs in a graph to maximize the spatial locality of GL requests as Algorithm 2 shows. Firstly, the reorganization algorithm selects the top Cn highest in-degree vertices with in-degree below the threshold Td as important vertices, where the threshold Td is used to exclude excessively high degree vertices since their neighborhood footprint often outsize the flash page and their locality can hardly be exploited. After that, it fetches each important vertex’s h hop neighbors as its working set. To reduce the complexity, the algorithm usually randomly samples a subset of the true working set (i.e.

Table 4: FPGA Resource Usage

Module	LUT	FF	BRAM	DSP
Flash Controller	44141	30156	80	0
NVMe Interface	8586	11455	28	0
GLA Accelerator	66287	51527	172	514
In Total	136506	117261	293	514
Percent(%)	62.45	26.82	53.76	57.11

\sqrt{N} from N vertices in our implementation) to represent the whole set. Then the important vertices are sorted according to the size of overlapping working set with others so that the potential spatial locality associated to the vertices are kept in the vertex sequence. Finally, the chosen important vertices and their corresponding h hop neighbors are assigned new IDs in sequence.

A tiny example shown in Figure 5 illustrates the graph reorganization procedure with given parameters: $h = 1$, $Cn = 3$, $Td = 0$. The procedure chooses three important vertices: $V5$, $V2$, and $V1$ according to the number of adjacent vertices and their one-hop neighbors are recorded as working-set respectively, as the shades shown in Figure 5. Then the algorithm sorts the three important vertices according to the size of overlapping working-set and obtains the sequence: $V5 \rightarrow V2 \rightarrow V1$ ($V5$ ’s working set has three common vertices with $V2$ ’s, and $V2$ ’s working set has two common vertices with $V1$ ’s). After that, each important vertex and the corresponding working set are assigned new IDs in the previously sorted order. Specifically, $V5$ and its five one-hop neighbors $V0$, $V4$, $V2$, $V7$, and $V9$ are assigned new IDs: $V0 \sim V5$. Then $V2$ and $V1$ follows. Finally, the procedure finds the remaining vertex that does not belong to any working set ($V6$) and assigns new ID to it to make sure that all the vertices in the graph are re-indexed.

5 Evaluation

5.1 GLIST Overall Evaluation

Experiment Setup. The Cosmos Plus OpenSSD platform was employed for the proposed GLIST system implementation, and it consists of an XC7Z045 FPGA chip (ARM-FPGA), 1 GB DRAM, an 8-channel NAND flash interface, an Ethernet interface, and a PCIe Gen2 8-lane interface. We implemented the GLA with Chisel [2] and integrated it in the hybrid ARM-FPGA processor as the major GL processing engine. The hardware project was synthesized and implemented with Vivado 2016.2 and the design works at 150MHz. Table 4 shows the logic resource usage of our hardware project. The firmware of the prototype runs on Dual 1GHz ARM Cortex-A9 core of XC7Z045. The board was connected with the host server via a PCIe link. We also profiled the prototype system and built a simulator for scalable evaluation.

We take a set of **Node-level**, **Edge-level**, and **Graph-level** GL workloads shown in Table 2 as benchmarks. The models used for benchmark are all quantized to 8bit fixed point. We

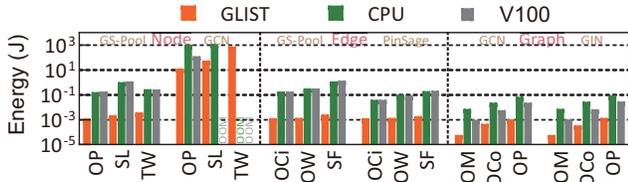


Figure 6: Energy consumption of a single request on different GL systems.

have the benchmark implemented with GLIST on Cosmos Plus OpenSSD platform to gain insight into the advantages of the in-storage graph learning. We compared GLIST with DGL [45] on a CPU-based platform and a GPU-based platform respectively. The CPU-based platform is equipped with two Intel Xeon E5-2690 V3 processors and 64 GB DRAM. The GPU-based platform includes two Intel Xeon E5-2690 V3 processors and an NVIDIA V100 GPU. Both platforms have all the graphs and GNN models initially stored in a Samsung 970 EVO 1 TB SSD with 3.5 GB/s peak read bandwidth because the large graphs used in many graph learning applications can exceed the capacity of the main memory. To evaluate the different systems, we randomly generated 10,000 graph learning requests over the graph and measured the average processing latency and energy consumption.

Performance. The performance of the proposed GLIST-based GL system is illustrated in Figure 7. It shows $13.2\times$ and $10.1\times$ average speedup compared to the CPU baseline and GPU baseline, respectively. Particularly, GLIST shows significant higher performance speedup on GS-Pool and PinSage which need to sample over the large input graphs. The main reason is that the random sampling over large input graphs incurs substantial random accesses to the flash and rather low flash bandwidth utilization when GS-Pool and PinSage are deployed on the CPU platform and the GPU platform. We also measured the flash bandwidth, and it shows only 100 MB/s, which is much lower than the peak bandwidth of the device and dramatically bottlenecks the computing capability of CPUs and GPUs accordingly. As a result, the performance of the CPU platform and the GPU platform is also similar. In contrast, GLIST with intensive data layout optimization and intra-request reuse optimization greatly improves the data reuse and reduces the random accesses over the flash. Thereby, it benefits most on GS-Pool and PinSage. Different from GS-Pool and PinSage, GCN and GIN operate on the entire graph instead of sampling sub-graphs. In this case, the graph will be accessed sequentially and the flash bandwidth can be fully utilized. With sufficient data supply from the flash, the GPU platform with more parallel processing engines shows much higher performance over the CPU platform according to the experiment. GLIST takes advantage of the specialized accelerator and still outperforms the CPU platform and the GPU platform given the same flash bandwidth provision.

Energy Consumption. In this experiment, we utilized a power meter to measure the power consumption of the pro-

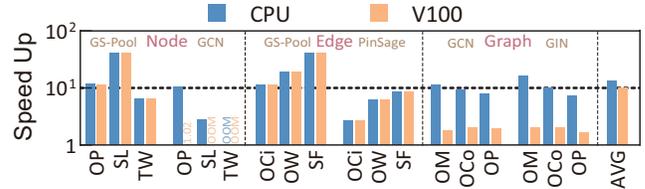


Figure 7: Single node performance of GLIST

posed GLIST system, the CPU-based, and the GPU-based graph learning systems respectively. Then, we obtained the energy consumption by calculating the production of the average power got by power meter and the benchmark execution time. The resulting energy consumption of the different benchmark GNN models are illustrated in Figure 6 and the per-request average power and benchmark time of different settings are listed in Table 5. It shows that GLIST reduces the average energy consumption by 98.7% and 98.0% respectively when compared to the CPU-based platform and the GPU-based platform. The significant energy reduction can be attributed to both the lower power consumption brought by the dedicated GLA in GLIST and the much lower execution time of GLIST as discussed in prior subsection. At the same time, we also noticed that the GPU-based platform shows higher energy consumption on PinSage and GS-Pool over the CPU-based platform. This is mainly caused by the fact that GPU fails to exploit its massive parallel processing engines due to the massive random access induced flash bandwidth bottleneck and much higher power consumption over the CPU-based platform. When the flash bandwidth utilization is improved for GCN and GIN that include more sequential data accesses, the execution time dominates the energy consumption in the GPU-based platform. Hence, the GPU-based platform exhibits lower energy consumption in these cases.

5.2 The GLIST Optimizations

Experimental Setup. To gain insight into the advantages of the GLIST optimization including graph reorganization (R), request scheduling (S), request grouping (G), and caching (C), we conducted a request generation server to continuously issue different graph analysis requests to the GLIST system for evaluating the above optimization strategies. In order to make the distribution of requests issued by generation servers closer to the real production system, we first analyze the real-world request trace from the commercial data center. The analysis results indicate that the requests have different levels of locality depending on the services provided by the data center and the data types accommodated in the warehouse nodes. Thereby, for simplicity, we introduce the $N\%$ -Locality, denoted as $N-L$, to describe the degree of locality in between the batch of requests that arrives in a fixed time window sent to GLIST. This term represents the $N\%$ neighbor vertices and edges can be reused between any adjacent request on average,

Table 5: Per-request average power and benchmark time of different platforms.

Dataset Model	Node						Edge						Graph					
	OP	SL	TW	SL	TW	OCi	OW	SF	OCi	OW	SF	OM	OCo	OP	OM	OCo	OP	
	GS-Pool			GCN			GS-Pool			PinSage			GCN		GIN			
$P_{GLIST}(W)$	25	25	25	26	25	26	25	24	25	26	24	25	25	26	25	26	25	25
$T_{GLIST}(ms)$	0.05	0.09	0.16	497.25	2252.24	30060.62	0.05	0.06	0.11	0.05	0.05	0.08	23.77	122.09	408.09	22.65	132.94	570.97
$P_{CPU}(W)$	280	281	280	201	200	-	311	282	280	292	202	288	282	202	219	202	202	219
$T_{CPU}(ms)$	0.60	3.80	1.03	5159.58	6191.28	-	0.58	1.17	4.37	0.13	0.33	0.68	0.027	0.11	0.32	0.037	0.13	0.42
$P_{GPU}(W)$	316	316	266	250	-	-	316	271	312	296	256	301	225	247	298	242	258	304
$T_{GPU}(ms)$	0.58	3.79	1.03	506.54	-	-	0.58	1.17	4.37	0.13	0.33	0.68	0.0043	0.024	0.079	0.0045	0.027	0.096

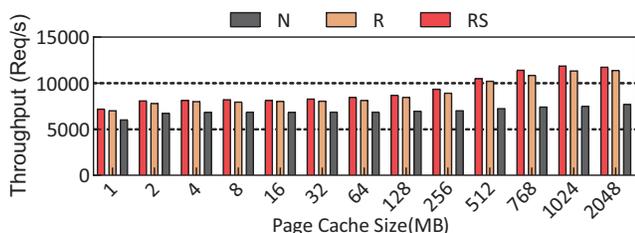


Figure 8: Throughput of GLIST w.r.t. page cache size.

which is defined as the follows:

$$N-L = \frac{N_{Common\ vertices\ of\ subgraph\ related\ to\ the\ two\ requests}}{N_{Vertices\ in\ subgraph\ related\ to\ the\ latter\ request}} \quad (7)$$

We randomly generated 10,000 vertex classification requests on the ogbn-papers100M dataset, whose property data is over 50 GB in size. We shuffled the requests to evaluate the advantages of the above optimization approaches. First, as discussed in Section 4.2, the benefits brought by the graph reorganization (**R**) and request scheduling (**S**) optimization methods are impacted by the cache size and the feature dimension. Thereby, in order to evaluate the influence of these factors on system performance, we evaluate the throughput of the GLIST system under different Page Cache sizes. After that, we adjusted the dimension of the vertex property data to distinguish the gains brought by the graph reorganization and request scheduling optimizations. Second, due to the locality level that can influence the performance of request scheduling, we fixed the property size of the vertices to 16 KB and adjust the locality level of the generated requests to show the variation of the gain of the request scheduling. Third, we explored the performance variation of the GLIST system under the different group size configurations of the GLIST runtime. Finally, we fixed the Page Cache size and measured the performance speedup of the GLIST system under different combinations of optimization methods compared to the GLIST system without optimization methods.

Evaluation. Figure 8 shows the throughput of the GLIST system under the three configurations including GLIST without optimization (**N**), GLIST with graph reorganization (**R**), and GLIST with graph reorganization and request scheduling (**RS**). It can be observed that the throughput of the system with **RS** and **R** increases as the Page Cache size increases because larger cache size can avoid vertex access being dispatched to flash memory and reduces access latency. Meanwhile, the GLIST system that adopts reorganization methods **R** pos-

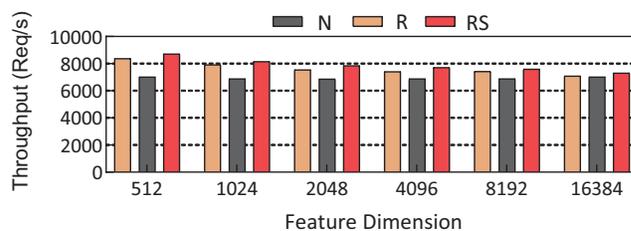


Figure 9: Throughput of GLIST w.r.t. property dimension.

sesses higher spatial data locality, which highly exploits the data reusability in each flash pages and alleviates the penalty of Page Cache misses, thereby, it performs better than the system without any optimizations, though the Page Cache is large. Furthermore, when both the request scheduling and graph reorganization are adopted in **RS** configuration, the spatial data locality of each single request can be exploited and released. In this case, as shown in Figure 8, the GLIST system with **RS** methods gains the highest throughput compared to **N** and **R** settings.

Figure 9 illustrates the system throughput of the GLIST system under the three configurations mentioned in the above paragraph with respect to various configurations on the property dimension. It can be observed that the GLIST with **R** and **RS** optimization methods achieve 19.5% and 24.2% higher throughput compared to the system without optimization **N**, respectively. As the feature dimension increases, a flash page can only accommodate a few property vectors, which results in the graph reorganization methods fails to exploit spatial data locality. Thereby, the system performance with **R** and **RS** optimization methods drops sharply and the **RS** still outperforms **R** because of the gain brought by request scheduling method. In addition, as the dimension of property vector increases, the performance gap between the **R** and **N** optimization methods gradually disappears until the property dimension arrives at the size of a flash page (16 KB). In this case, the system with request scheduling still maintains 4% higher throughput than the other settings.

Figure 10 (a) shows the speedup of the GLIST system with request scheduling methods **RS** under different locality level configuration compared to a system without optimization methods. The increase of locality level indicates the rise of data reusability, thus the GLIST system adopting request scheduling method achieves performance improvement up to 2.65 \times . As shown in Figure 10 (b), the GLIST system achieves performance improvement compared to the system without

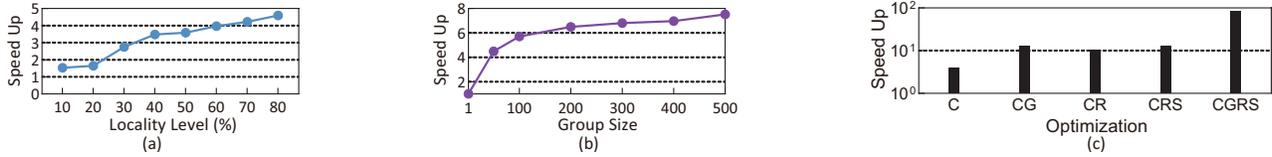


Figure 10: (a) Speedup of request scheduling w.r.t. locality level. (b) Performance improvement of different group size in request grouping. (c) Performance improvement of different optimization enabled.

request grouping optimization with the increase of group size. The reason can be attributed to two folders: (1) larger group size can fully utilize the internal bandwidth provided by multiple flash channels; (2) large group size can fully exploit the data parallelism and thus making the computation unit of the graph learning accelerator in high utilization. In addition, the limitation of internal flash bandwidth makes performance improvement slow down when the group size is larger than 200. Figure 10 (c) shows the performance speedup of the GLIST system under the combination of various optimization (R, S, G, and C) compared to the system without optimization (N). It can be observed that neither the combination of node embedding caching and request grouping ($12.94\times$), denoted as CG, nor the combination of node embedding caching, graph reorganization, and request scheduling ($12.89\times$), denoted as CRS achieves the best performance. This is because CG fails to exploit the data locality brought the graph reorganization and request scheduling methods under the high utilization of flash bandwidth and graph learning accelerator, which results in the Page Cache hit rate only reaches to 67.86%. Meanwhile, although CRS can exploit the data locality using graph organization and make Page Cache hit rate reach to 94.73%. CRS is unable to fully utilize the flash bandwidth and PE-Array in GLA. Not only does the CGRS optimization method exploit the data locality but also fully utilize the available resource, making the GLIST system achieve the highest performance improvement.

5.3 Bit-width Scalability Exploration

The bit-width of vertex feature vector and GNN model parameters has a great impact on performance, resource overhead, and energy consumption of the GLIST prototype. To explore an appropriate setting for quantization, we choose GCN [21] and Cora [39] as target model and dataset respectively to evaluate how the four configurations including **floating-point, 32 bit fixed-point, 16 bit fixed point, and 8 bit fixed-point** impact on accuracy, latency, energy consumption, and resource usage. We leverage a static quantization method which enumerates every possible configuration at each layer and choose the best one with the lowest loss. The results are as shown in Table 6. Though **floating-point** and **32 bit fixed-point** achieve higher accuracy, the logic resource usage is extremely high and can hardly be implemented on current FPGA platform. And wider word size makes it hard to enable high-throughput graph learning services because of high bandwidth

Table 6: Comparison of accuracy and resource utilization

Config.	Acc.	LUT	FF	DSP	Latency	Energy
float-32	79.1%	1891986	192372	1	-	-
fixed-32	77.8%	632777	242506	889	-	-
fixed-16	77.5%	125253	112730	513	2.74ms	$5.7\times 10^{-2}J$
fixed-8	77.1%	66287	51527	514	1.80ms	$3.6\times 10^{-2}J$

requirements. For the lower bit configurations, the latency decreases by 34.3% and the energy consumption decreases by 36.8% with only a 0.4% loss on accuracy when changing bit width from 16 to 8. Moreover, the resource usage of 8-bit GLA is significantly less than another one, making it possible to implement more GLA cores in the GLIST system to perform higher throughput graph learning services. Note that the quantization method used for evaluation is only a naive one and the accuracy of low-bit configurations will show better results when changing to state-of-the-art methods [9, 12, 13].

6 Conclusion

In this paper, we formulated that the conventional GPU+SSD graph learning platforms are limited by I/O operations after studying a diverse set of graph learning tasks. We then study the data locality that exists in flash-based graph learning applications. To tackle the bottlenecks of conventional graph learning systems, we proposed an in-storage graph learning accelerating system, GLIST, which features multiple optimizations proposed based on our observations to fully exploit data locality. Finally, we implemented a GLIST prototype with FPGA and showed it achieves $13.2\times$ and $10.1\times$ average speedup and reduces the power consumption by 98.7%, 98.0% compared to conventional CPU and GPU based graph learning systems, respectively.

7 Acknowledgement

We are grateful to Professor Zsolt István for his useful comments and suggestions on improving this paper. This work is supported in part by the National Key Research and Development Program of China under grant 2018AAA0102705, and in part by the National Natural Science Foundation of China (NSFC) under grant No.(62090024, 61876173, 61902375). The corresponding authors are Ying Wang and Cheng Liu.

References

- [1] The OpenSSD Project. <http://openssd.io>.
- [2] Jonathan Bachrach, Huy Vo, Brian Richards, Yun-sup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.
- [3] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54, 2006.
- [4] Bradley R Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Mike Personick, Karthik Rajan, et al. Amazon neptune: Graph data management in the cloud. In *International Semantic Web Conference (P&D/Industry/BlueSky)*, 2018.
- [5] Fabian Beck, Michael Burch, Stephan Diehl, and Daniel Weiskopf. A taxonomy and survey of dynamic graph visualization. In *Computer Graphics Forum*, volume 36, pages 133–159. Wiley Online Library, 2017.
- [6] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.
- [7] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.
- [8] Chanwoo Chung, Jinhyung Koo, Junsu Im, and Sungjin Lee. Lightstore: Software-defined network-attached key-value drives. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 939–953, 2019.
- [9] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *arXiv preprint arXiv:1511.00363*, 2015.
- [10] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.
- [11] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moon-sang Kwon, Chanho Yoon, Sangyeun Cho, et al. Biscuit: A framework for near-data processing of big data workloads. *ACM SIGARCH Computer Architecture News*, 44(3):153–165, 2016.
- [12] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented approximation of convolutional neural networks. *arXiv preprint arXiv:1604.03168*, 2016.
- [13] Philipp Gysel, Jon Pimentel, Mohammad Motamedi, and Soheil Ghiasi. Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks. *IEEE transactions on neural networks and learning systems*, 29(11):5784–5789, 2018.
- [14] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.
- [15] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.
- [16] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.
- [17] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. Yoursql: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment*, 9(12):924–935, 2016.
- [18] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, et al. Grafboost: Using accelerated flash storage for external graph analytics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 411–424. IEEE, 2018.
- [19] Kristian Kersting, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann. Benchmark data sets for graph kernels, 2016. <http://graphkernels.cs.tu-dortmund.de>.
- [20] Minsub Kim and Sungjin Lee. Reducing tail latency of dnn-based recommender systems using in-storage processing. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 90–97, 2020.
- [21] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.
- [22] Pradeep Kumar and H Howie Huang. G-store: high-performance graph store for trillion-edge processing. In

SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 830–841. IEEE, 2016.

- [23] Pradeep Kumar and H. Howie Huang. Graphone: A data store for real-time analytics on evolving graphs. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 249–263, Boston, MA, February 2019. USENIX Association.
- [24] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.
- [25] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoun Choi, H Peter Hofstee, Gi-Joon Nam, Mark R Nutter, and Damir Jamsek. Extrav: boosting graph processing near storage with a coherent accelerator. *Proceedings of the VLDB Endowment*, 10(12):1706–1717, 2017.
- [26] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>, June 2014.
- [27] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, LI Huawei, Dawen Xu, and Xiaowei Li. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks. *IEEE Transactions on Computers*, 2020.
- [28] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A deep learning engine for in-storage data retrieval. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 395–410, Renton, WA, July 2019. USENIX Association.
- [29] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *Internet Computing, IEEE*, 7(1):76–80, 2003.
- [30] Hang Liu and H. Howie Huang. Graphene: Fine-grained IO management for graph computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 285–300, Santa Clara, CA, February 2017. USENIX Association.
- [31] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 527–543, 2017.
- [32] Vikram Sharma Mailthody, Zaid Qureshi, Weixin Liang, Ziyang Feng, Simon Garcia De Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wen-mei Hwu. Deepstore: In-storage acceleration for intelligent queries. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 224–238, 2019.
- [33] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.
- [34] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. Graphssd: graph semantics aware ssd. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 116–128, 2019.
- [35] Heiko Paulheim. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic web*, 8(3):489–508, 2017.
- [36] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488, 2013.
- [37] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing in-storage computing system for emerging high-performance drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, Renton, WA, July 2019. USENIX Association.
- [38] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pages 593–607. Springer, 2018.
- [39] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.
- [40] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(77):2539–2561, 2011.
- [41] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Graphr: Accelerating graph processing using rram. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 531–543. IEEE, 2018.

- [42] Aravind Subramanian, Pablo Tamayo, Vamsi K Mootha, Sayan Mukherjee, Benjamin L Ebert, Michael A Gillette, Amanda Paulovich, Scott L Pomeroy, Todd R Golub, Eric S Lander, et al. Gene set enrichment analysis: a knowledge-based approach for interpreting genome-wide expression profiles. *Proceedings of the National Academy of Sciences*, 102(43):15545–15550, 2005.
- [43] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Bin-qiang 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] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909.01315*, 2019.
- [45] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.
- [46] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. Knowledge graph embedding: A survey of approaches and applications. *IEEE Transactions on Knowledge and Data Engineering*, 29(12):2724–2743, 2017.
- [47] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.
- [48] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.
- [49] Mingyu Yan, Zhaodong Chen, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Characterizing and understanding gcns on gpu. *IEEE Computer Architecture Letters*, 2020.
- [50] Pinar Yanardag and SVN Vishwanathan. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1365–1374, 2015.
- [51] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.
- [52] Hao Yin, Austin R Benson, Jure Leskovec, and David F Gleich. Local higher-order graph clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 555–564, 2017.
- [53] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 974–983, 2018.
- [54] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
- [55] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018.
- [56] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 45–58, 2015.
- [57] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, Santa Clara, CA, February 2015. USENIX Association.
- [58] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment*, 12(12):2094–2105, 2019.

DART: A Scalable and Adaptive Edge Stream Processing Engine

Pinchao Liu
Florida International University

Dilma Da Silva
Texas A&M University

Liting Hu*
Virginia Tech

Abstract

Many Internet of Things (IoT) applications are time-critical and dynamically changing. However, traditional data processing systems (e.g., stream processing systems, cloud-based IoT data processing systems, wide-area data analytics systems) are not well-suited for these IoT applications. These systems often do not scale well with a large number of concurrently running IoT applications, do not support low-latency processing under limited computing resources, and do not adapt to the level of heterogeneity and dynamicity commonly present at edge environments. This suggests a need for a new edge stream processing system that advances the stream processing paradigm to achieve efficiency and flexibility under the constraints presented by edge computing architectures.

We present DART, a scalable and adaptive edge stream processing engine that enables fast processing of a large number of concurrent running IoT applications' queries in dynamic edge environments. The novelty of our work is the introduction of a dynamic dataflow abstraction by leveraging distributed hash table (DHT) based peer-to-peer (P2P) overlay networks, which can automatically place, chain, and scale stream operators to reduce query latency, adapt to edge dynamics, and recover from failures.

We show analytically and empirically that DART outperforms Storm and EdgeWise on query latency and significantly improves scalability and adaptability when processing a large number of real-world IoT stream applications' queries. DART significantly reduces application deployment setup times, becoming the first streaming engine to support DevOps for IoT applications on edge platforms.

1 Introduction

Internet-of-Things (IoT) applications such as self-driving cars, interactive gaming, and event monitoring have a tremendous potential to improve our lives. These applications generate a large influx of sensor data at massive scales (millions of sensors, hundreds of thousands of events per second [20,26]). Under many time-critical scenarios, these massive data streams must be processed in a *very short time* to derive actionable intelligence. However, many IoT applications [22,23] adopt the server-client architecture, where the front-end sensors send time-series observations of the physical or human system

*Liting is affiliated with Virginia Tech, but was at Florida International University during this work.

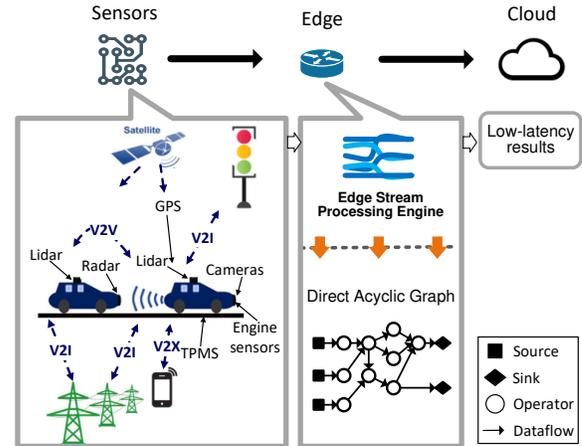


Figure 1: Edge stream processing use case.

to the back-end cloud for analysis. Such a long-distance of processing makes it *not* appropriate or time-critical IoT application because: (1) the high latency may cause the results to be obsolete; and (2) the network infrastructure cannot afford the massive data streams.

A new trend to address this issue is edge stream processing. To put it simply, edge stream processing applies the stream processing paradigm to the edge computing architecture [37,50]. Instead of relying on the cloud to process sensor data, the edge stream processing system relies on distributed edge compute nodes (Gateways, edge routers, and powerful sensors) which are near the data sources to process data and trigger actuators. The execution pipeline is as follows. Sensors (e.g., self-driving car sensors, smart wearables) generate data streams continuously. They are then consumed by the edge stream processing engine, which creates a logical topology of stream processing operators connected into a Directed Acyclic Graph (DAG), processes the tuples of streams as they flow through the DAG from sources to sinks, and outputs the results in a very short time. Each *source* node is an IoT sensor. Each *inner* node runs an operator or operators that can perform user-defined computation on data, ranging from simple computation such as *map*, *reduce*, *join*, *filter* to complex computation such as ML-based classification algorithms. Each *sink* node is an IoT actuator or a message queue to the cloud.

Figure 1 illustrates a use case scenario [10] that benefits from an edge stream processing engine. In future Intelligent Transportation Systems such as the efforts currently funded

by the US Department of Transportation [27], cars are interconnected and equipped with wide-area network access. Even at low levels of autonomy, each car will generate at least 3 Gbit/s of sensor data [25]. On the back-end, many IoT stream applications will run concurrently, consuming these live data streams to quickly derive insights and make decisions. Examples of such applications include peer-to-peer services for traffic control and car-sharing safety and surveillance systems. Note that many of these services cannot be completed on on-board computers within a single car, requiring the cooperation of many computers, edge routers, and gateways with sensors and actuators as sources and sinks. They will involve a large number of cars and components from the road infrastructure.

However, as IoT systems grow in number and complexity, we face significant challenges in building edge stream processing engines that can meet their needs.

The first challenge is: *how to scale to numerous concurrently running IoT stream applications?* Due to the exponential growth of new IoT users, the number of concurrently running IoT stream applications will be significantly large and change dynamically. However, modern stream processing engines such as Storm [7], Flink [32], and Heron [44] and wide-area data analytic systems [39–41, 43, 53, 55, 62, 65, 66] mostly inherit a centralized architecture, in which the monolithic master is responsible for all scheduling activities. They use a first-come, first-serve method, making deployment times accumulate and leading to long-tail latencies. As such, this centralized architecture easily becomes scalability and performance bottlenecks.

The second challenge is: *how to adapt to the edge dynamics and recover from failures to ensure system reliability?* IoT stream applications run in a highly dynamic environment with load spikes and unpredictable occurrences of events. Existing studies on the adaptability in stream processing systems [34, 36, 38, 42, 64] mainly focus on the cloud environment, where the primary sources of dynamics come from workload variability, failures, and stragglers. In this case, a solution typically allocates additional computational resources or re-distributes the workload of the bottleneck execution across multiple nodes within a data center. However, the edge environment imposes additional difficulties: (1) edge nodes leave or fail unexpectedly (e.g., due to signal attenuation, interference, and wireless channel contention); and (2) accordingly, stream operators fail more frequently. Unfortunately, unlike the cloud servers, edge nodes have limited computing resources: few-core processors, little memory, and little permanent storage [37, 59] and they have no backpressure. As such, the previous adaptability techniques by re-allocating resources or buffering data at data sources cannot be applied in edge stream processing systems.

We present DART, a scalable and adaptive edge stream processing engine to address the challenges listed above. The key innovation is that DART re-architects the stream processing system runtime design. In sharp contrast to existing stream

processing systems, there is no monolithic master. Instead, DART involves all peer nodes to participate in operator placement, dataflow path planning, and operator scaling, thereby revolutionarily improving scalability and adaptivity.

We make the following contributions in this paper.

First, we study the software architecture of existing stream processing systems and discuss their limitations in the edge setting. To our best knowledge, we are the first to observe the lack of scalability and adaptivity in stream processing systems for handling a large number of IoT applications (Sec. 2).

Second, we design a novel dynamic dataflow abstraction to automatically place, chain and parallelize stream operators using the distributed hash table (DHT) based peer-to-peer (P2P) overlay networks. The main advantage of a DHT is that it avoids the original monolithic master. All peer nodes jointly make operator-mapping decisions. Nodes can be added or removed with minimal work around re-distributing keys. This design allows our system to scale to extremely large numbers of nodes. To our best knowledge, we are the first to explore DHTs to pursue extreme scalability in edge stream processing (Sec. 3).

Third, using DHTs, we decompose the stream processing system architecture from $1:n$ to $m:n$, which removes the centralized master and ensures that each edge zone can have an independent master for handling applications and operating autonomously without any centralized state (Sec. 4). As a result of our distributed management, DART improves overall query latencies for concurrently executing applications and significantly reduces application deployment times. To the best of our knowledge, we offer the first stream processing engine to make it feasible to operate IoT applications in a DevOps fashion.

Finally, We demonstrate DART’s scalability and latency gains over Apache Storm [7] and EdgeWise [1] on IoT stream benchmarks (Sec. 5).

2 Background

2.1 Stream Processing Programming Model

Data engineers define an IoT stream application as a directed acyclic graph (DAG) that consists of operators (see Figure 1). Operators run user-defined functions such as `map`, `reduce`, `join`, `filter`, and ML algorithms. Data tuples flow through operators along the DAG (topology). In our case, DART supports both stateful batch processing by using *windows* as well as continuously event-based stateless processing. The application’s query latency is defined as the elapsed time since the source operator receives the timestamp signaling the completion of the current window to when the sink operator externalizes the window results.

We consider typical edge environments. The edge compute nodes consist of sensors, routers, and sometimes gateways. They are connected by different connections such as WiFi,

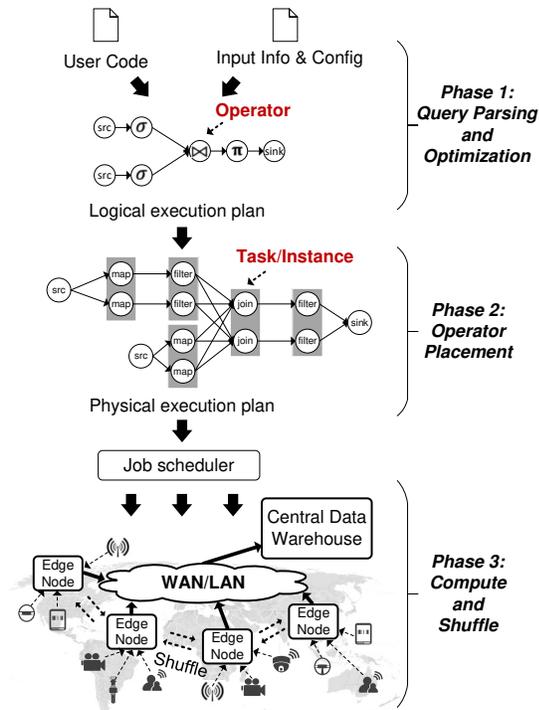


Figure 2: IoT stream applications execution pipeline.

Zigbee, BlueTooth, or LAN with diverse inbound and outbound bandwidths and latency. They have fewer resources compared to the cloud servers, but more resources than embedded sensor networks, and thus can afford reasonably complex operations (e.g., SenML parsers, Kalman filters, linear regressions). As shown in Figure 2, the execution pipeline for processing an IoT stream application has a few key phases:

- **Phase 1: Query parsing and optimization.** When an IoT stream application is submitted by a user, its user code containing transformations and actions is first parsed into a logical execution plan represented using a DAG, where the vertices correspond to stream operators and the edges refer to data flows between operators.
- **Phase 2: Operator placement.** Afterward, the DAG is converted into a physical execution plan, which consists of several execution stages. Each stage can be further broken down into multiple execution instances (tasks) that run in parallel, as determined by the stage’s level of parallelism. This requires the system to place all operators’ instances on distributed edge nodes that can minimize the query latency and maximize the throughput.
- **Phase 3: Compute and shuffle.** Operator instances independently compute their local shard of data and shuffle the intermediate results from one stage to the next stage. This requires the system to adapt to the workload variations, bandwidth variations, node joins and leaves, and failures and stragglers.

2.2 Stream Processing System Architecture

As shown in Figure 3, existing studies [37, 39–41, 43, 50, 53, 55, 62, 65, 66] mostly rely on a master-slave architecture, in which a “single” monolithic master is administering many applications (if any). The responsibilities include accepting new applications, parsing each application’s DAG into stages, determining the number of parallel execution instances (tasks) under each stage, mapping these instances onto edge nodes, and tracking their progress.

This centralized architecture may run well for handling a small number of applications in the cloud. However, when it comes to IoT systems in the edge environment, new IoT users join and exit more frequently and launch a large number of IoT applications running at the same time, which makes the architecture easily become a scalability bottleneck and jeopardize the application’s performance. This is because of (1) **high deployment latency**. These systems use a first-come, first-served approach to deploy applications, which causes applications to wait in a long queue and thus leads to long query latencies; and (2) **lack of flexibility for dataflow path planning**. They limit themselves to a fixed execution model and lack the flexibility to design different dataflow paths for different applications to adapt to the edge dynamics.

The limitation of the centralized architecture has been identified before in data processing frameworks such as YARN [63], Sparrow [51], Apollo [30]. They use two masters for task scheduling (one is the main master and one is the backup master). However, they remain fundamentally centralized [30, 51, 63] and restrict themselves to handle a small number of applications only.

3 Design

This section introduces DART’s dynamic dataflow abstraction and shows how to scale up and down operators and perform failure recovery on top of this abstraction.

3.1 Overview

The DART system aims to achieve the following goals:

- **Low latency.** It achieves low latency for IoT queries.
- **Scalability.** It can process a large number of concurrently running applications at the same time.
- **Adaptivity.** It can adapt to the edge dynamics and recover from failures.

As shown in Figure 4, DART consists of three layers: *the DHT-based consistent ring overlay*, *the dynamic dataflow abstraction*, and *the scaling and failure recovery mechanisms*.

Layer 1: DHT-based consistent ring overlay. All distributed edge “nodes” (e.g., routers, gateways, or powerful sensors) are self-organized into a DHT-based overlay, which has been commonly used Bitcoin [48] and BitTorrent [35].

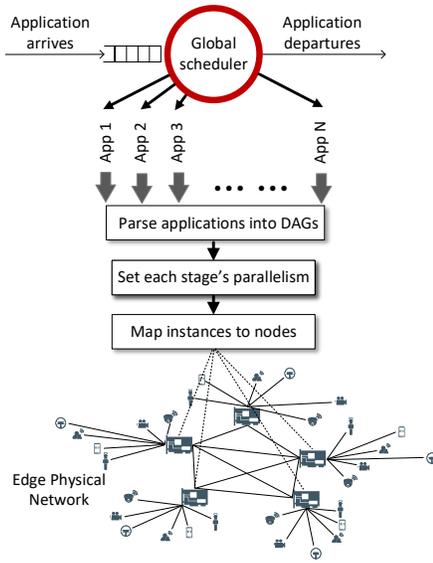


Figure 3: The global scheduler.

Each node is randomly assigned a unique “NodeID” in a large circular NodeID space. NodeIDs are used to identify the nodes and route stream data. *No matter where the data is generated, it is guaranteed that the data can be routed to any destination node within $O(\log N)$ hops.* To do that, each node needs to maintain two data structures: a routing table and a leaf set. The routing table is used for building dynamic dataflows. The leaf set is used for scaling and failure recovery.

Layer 2: Dynamic dataflow abstraction. Built upon the overlay, we introduce a novel dynamic dataflow abstraction. The key innovation is to leverage DHT-based routing protocols to approximate the optimal routes between source nodes and sink nodes, which can automatically place and chain operators to form a dataflow graph for each application.

Layer 3: Scaling and failure recovery mechanisms. Every node has a leaf set that contains physically “closest” nodes to this node. The leaf set provides the elasticity for (1) scaling up and down operators to adapt to the workload variations; (2) re-planning dataflows to adapt to the network variations. As stream data moves along the dataflow graph, the system makes dynamic decisions about the downstream node to send streams to, which increases network path diversity and becomes more resilient to changes in network conditions; and (3) replicating operators to handle failures and stragglers. If any node fails or becomes a straggler, the system can automatically switch over to a replica.

3.2 Dynamic Dataflow Abstraction

In the P2P model (e.g., Pastry [57], Chord [61]), each node is equal to the other nodes, and they have the same rights and duties. The primary purpose of the P2P model is to enable

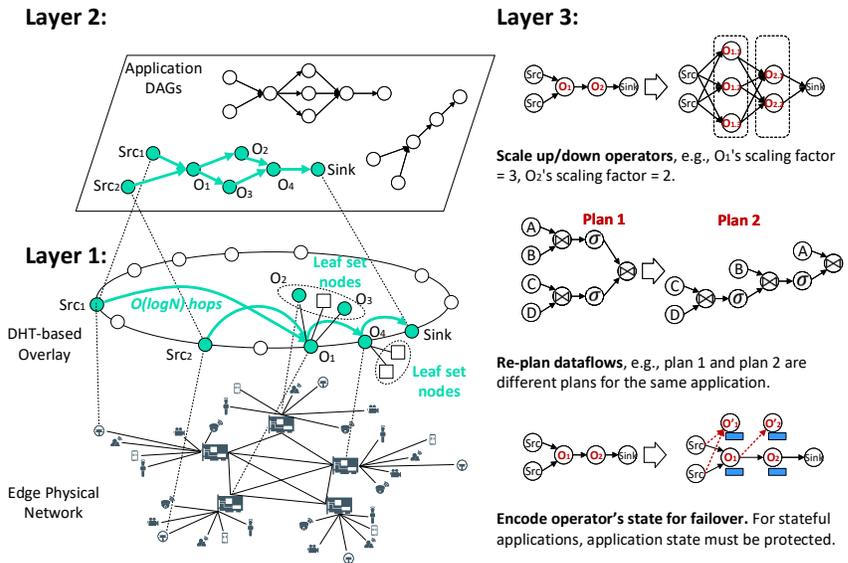


Figure 4: Dynamic dataflow graph abstraction for operator placement.

all nodes to work collaboratively to deliver a specific service. For example, in BitTorrent [35], if someone downloads some file, the file is downloaded to her computer in bits and parts that come from many other computers in the system that already have that file. At the same time, the file is also sent (uploaded) from her computer to others who ask for it. Similar to BitTorrent in which many machines work collaboratively to undertake the duties of downloading and uploading files, we enable all distributed edge nodes to work collaboratively to undertake the duties of the original monolithic master’s.

Figure 5 shows the process of building the dynamic dataflow graph for an IoT stream application. *First*, we organize distributed edge nodes into a P2P overlay network, which is similar to the BitTorrent nodes that use the Kademila DHT [46] for “trackerless” torrents. Each node is randomly assigned a unique identifier known as the “NodeID” in a large circular node ID space (e.g., $0 \sim 2^{128}$). *Second*, given a stream application, we map the source operators to the sensors that generate the data streams. We map the sink operators to IoT actuators or message queues to the Cloud service. *Third*, every source node sends a JOIN message towards a key, where the key is the hash of the sink node’s NodeID. Because all source nodes belonging to the same application have the same key, their messages will be routed to a rendezvous point—the sink node(s). Then we keep a record of the nodes that these messages pass through during routings and link them together to form the dataflow graph for this application.

To achieve low latency, the overlay guarantees that the stream data can be routed from source nodes to sink nodes within $O(\log N)$ hops, thus ensuring the query latency upper bound. To achieve locality, the dynamic dataflow graph covers a set of nodes from sources to sinks. The first hop is always

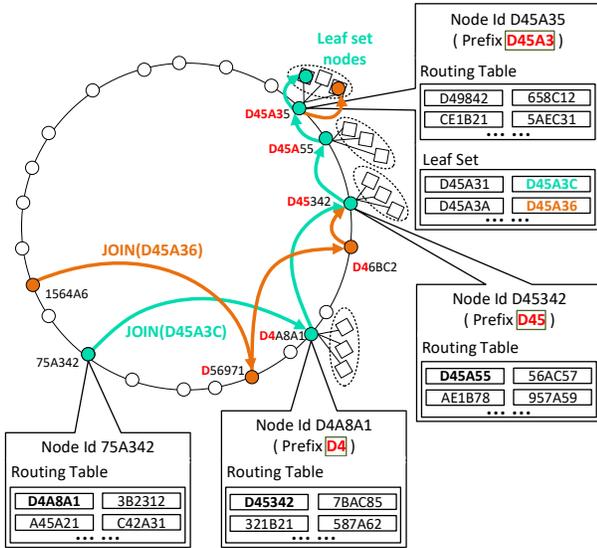


Figure 5: The process of building dynamic dataflow graph.

the node closer to the data source (data locality). Each node in the path has many leaf set nodes, which provides enough heterogeneous candidate nodes with different capacities and increases network path diversity. For example, if there are more operators than nodes, extra operators can map onto leaf set nodes. For that purpose, each node maintains two data structures: a routing table and a leaf set.

- *Routing table*: it consists of node characteristics organized in rows by the length of the common prefix. The routing works based on prefix-based matching. Every node knows m other nodes in the ring and the distance of the nodes it knows increases exponentially. It jumps closer and closer to the destination, like a greedy algorithm, within $\lceil \log_{2b} N \rceil$ hops. We add extra entries in the routing table to incorporate proximity metrics (e.g., hop count, RTT, cross-site link congestion level) in the routing process so as to handle the bandwidth variations.
- *Leaf set*: it contains a fixed number of nodes whose NodeIds are “physically” closest to that node, which assists in rebuilding routing tables and reconstructing the operator’s state when any node fails.

As shown in Figure 5, node 75A342 and node 1564A6 are two source nodes and node D45A3C is the sink node. The source nodes route JOIN messages towards the sink node, and their messages are routed to a rendezvous point (s) — the sink node(s). We choose the forwarder nodes along routing paths based on RTT and node capacity. Afterward, we keep a record of the nodes that their messages pass through during routings (e.g., node D4A8A1, node D45342, node D45A55, node D45A35, node D45A3C), and reversely link them together to build the dataflow graph.

The key to efficiency comes from several factors. First, the application’s instances can be instantly placed without

the intervention of any centralized master, which benefits the time-critical deadline-based IoT application’s queries. Second, because keys are different, the paths and the rendezvous nodes of all application’s dataflow graphs will also be different, distributing operators evenly over the overlay, which significantly improves the scalability. Third, the DHT-based leaf set increases elasticity for handling failures and adapting to the bandwidth and workload variations.

3.3 Elastic Scaling Mechanism

After an application’s operators are mapped onto the nodes along this application’s dataflow graph, *how to auto-scale them to adapt to the edge dynamics?* We need to consider various factors. Scaling up/down is to increase/decrease the parallelism (#instances) of the operator within a node. Scaling out is to instantiate new instances on another node by re-distributing the data streams across extra network links. In general, scaling up/down incurs smaller overhead. However, scaling out can solve the bandwidth bottleneck by increasing network path diversity, while scaling up/down may not.

We design a heuristic approach that adapts execution based on various factors. If there are computational bottlenecks, we scale up the problematic operators. The intuition is that when data queuing increases, automatically adding more instances to the system will avoid the bottleneck. We leverage the Secant root-finding method [29] to automatically calculate the optimal instance number based on the current system’s health value. The policy is pluggable. Let $f(x)$ represent the health score based on the input rate and the queue size ($0 < f(x) < 1$, with 1 being the highest score). Let x_n and x_{n-1} be the number of instances during phases p_n and p_{n-1} . Then the number of instances required for the next phase p_{n+1} such that $f \cong 1$ can be given by:

$$x_{n+1} = x_n + (1 - f(x_n)) \times \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \quad (1)$$

For bandwidth bottlenecks, we further consider whether the operator is stateless or stateful. In the case of stateless operators, we simply scale out operators across nodes. For stateful operators, we migrate the operator with its state to a new node in the leaf set that increases the network path diversity. Intuitively, when the original path only achieves low throughput, an operator may achieve higher throughput by sending the data over another network path.

3.4 Failure Recovery Mechanism

Since the overlay is self-organizing and self-repairing, the dataflow graph for each IoT application can be automatically recovered by restarting the failed operator on another node. Here, the challenge is, *how to resume the processing without losing intermediate data (i.e., operator state)?* Examples of operator states include keeping some aggregation or summary

of the received tuples in memory or keeping a state machine for detecting patterns for fraudulent financial transactions in memory. A general approach is checkpointing [7, 8, 54, 64], which periodically checkpoints all operators' states to a persistent storage system (e.g., HDFS) and the failover node retrieves the checkpointed state upon failures. This approach, however, is slow because it must transfer state over the edge networks that typically have very limited bandwidth.

We design a parallel recovery approach by leveraging the robustness of the P2P overlay and our previous experience in stateful stream processing [45]. Periodically, the larger-than-memory state is divided, replicated, and checkpointed to each node's leaf set nodes by using erasure codes [56]. Once any failure happens, the backup node takes over and retrieves state fragments from a subset of leaf set nodes to recompute state and resume processing. By doing that, we do not need a central master. The failure recovery process is fast because many nodes can leverage the dataflow graph to recompute the lost state in parallel upon failures. The replica number, the checkpointing frequency, the number of encoded blocks and the number of raw blocks are tunable parameters. They are determined based on state size, running environment and the application's service-level agreements (SLAs.)

4 Implementation

Instead of implementing another distributed system core, we implement DART on top of Apache Flume [3] (v.1.9.0) and Pastry [16] (v.2.1) software stacks. Flume is a distributed service for collecting and aggregating large amounts of streaming event data, which is widely used with Kafka [4] and the Spark ecosystem. Pastry is an overlay network and routing network for the implementation of a distributed hash table (DHT) similar to Chord [61], which is widely used in applications such as Bitcoin [48], BitTorrent [35], and FAROO [15]. We leverage Flume's excellent runtime system (e.g., basic API, code interpreter, transportation layer) and Pastry's routing substrate and event transport layer to implement the DART system.

We made three major modifications to Flume and Pastry: (1) we implemented the dynamic dataflow abstraction for operator placement and path planning algorithm, which includes a list of operations to track the DHT routing paths for chaining operators and a list of operations to capture the performance metrics of nodes for placing operators; (2) we implemented the scaling mechanism and the failure recovery mechanism by introducing queuing-related metrics (queue length, input rate, and output rate), buffering operator's in-memory state, encoding and replicating state to leaf set nodes; and (3) we implemented the distributed schedulers by using Scribe [33] topic-based trees on top of Pastry.

Figure 6 shows the high-level architecture of the DART system. The system has two components: a set of distributed schedulers that span geographical zones and a set of workers. Unlike traditional stream processing systems that manually

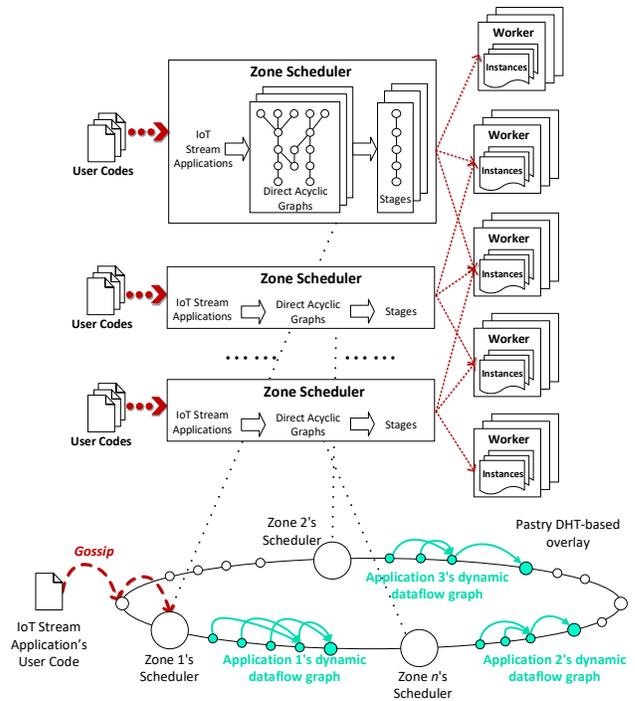


Figure 6: The DART system architecture.

assign nodes as “master” or “workers”, DART dynamically assigns nodes as “schedulers” or “workers”. For the first step, when any new IoT stream application is launched, it looks for a nearby scheduler by using the gossip protocol [31], which is a procedure of P2P communication that is based on the way that epidemics spread. If it successfully finds a scheduler within $\log(N)$ hops, the application registers itself to this scheduler. Otherwise, it votes any random nearby node to be the scheduler and registers itself to that scheduler. For the second step, the scheduler processes this application's queries by parsing the application's user code into a DAG and dividing this DAG into stages. Then the scheduler automatically parallelizes, chains operators, and places the instances on edge nodes using *the proposed dynamic dataflow abstraction*. These nodes are then set as this application's workers. The system automatically scales up and out operators, re-plans, and replicates operators to adapt to the edge dynamics and recover from failures by using *the proposed scaling mechanism and failure recovery mechanism*.

The key to efficiency comes from several factors. First, all nodes in the system are equal peers with the same rights and duties. Each node may act as one application's worker, another application's worker, a zone's scheduler, or any combination of the above, resulting in all load being evenly distributed. Second, the scheduler is no longer any central bottleneck. Third, the system automatically creates more schedulers for application intensive zones and fewer ones for sparse zones, thus scaling to extremely large numbers of nodes and applications.

5 Evaluation

We evaluate DART on a real hardware testbed (using Raspberry Pis) and emulation testbed in a distributed network environment. We explore its performance for real-world IoT stream applications. Our evaluation answers these questions:

- Does DART improve latency when processing a large number of IoT stream applications?
- Does DART scale with the number of concurrently running IoT stream applications?
- Does DART improve adaptivity in the presence of workload changes, transient failures and mobility?
- What is the runtime overhead of DART?

5.1 Setup

Real hardware. Real hardware experiments use an intermediate class computing device representative of IoT edge devices. Specifically, we use 10 Raspberry Pi 4 Model B devices for hosting source operators, each of which has a 1.5GHz 64-bit quad-core ARMv8 CPU with 4GB of RAM and runs Linux raspberrypi 4.19.57. Raspberry Pis are equipped with Gigabit Ethernet Dual-band Wi-Fi. We use 100 Linux virtual machines (VMs) to represent the gateways and routers for hosting internal and sink operators, each of which has a quad-core processor and 1GB of RAM (equivalent to Cisco’s IoT gateway [11]). These VMs are connected through a local-area network. In order to make our experiments closer to real edge network scenarios, we used the TC tool [17] to control link bandwidth differences.

Emulation deployment. Emulation experiments are conducted on a testbed of 100 VMs running Linux 3.10.0, all connected via Gigabit Ethernet. Each VM has 4 cores and 8GB of RAM, and 60GB disk. Specifically, to evaluate DART’s scalability, we use one JVM to emulate one logical edge node and can emulate up to 10,000 edge nodes in our testbed.

Baseline. We used Storm and EdgeWise [37] as the edge stream processing engine baseline. Apache Storm version is 2.0.0 [7] and EdgeWise [37] is downloaded from GitHub [14]. Both of them are configured with 10 TaskManagers, each with 4 slots (maximum parallelism per operator = 36). We run Nimbus and ZooKeeper [9] on the VMs and run supervisors on the Raspberry Pis. We use Pastry 2.1 [57] configured with leaf set size of 24, max open sockets of 5000 and transport buffer size of 6 MB.

Benchmark and applications. We deploy a large number of applications (topologies) simultaneously to demonstrate the scalability of our system. The applications in the mixed set are chosen from a full-stack standard IoT stream processing benchmark [60]. We also implement four IoT stream processing applications that use real-world datasets [12, 13, 24, 47]. They employ various techniques such as predictive analysis, model training, data preprocessing, and statistical summarization. Their operators run functions such as `transform`,

`filter`, `flatmap`, `aggregate`, `duplicate`, and `hash`. For example, we implement the DEBS 2015 application [13] to process spatio-temporal data streams and calculate real-time indicators of the most frequent routes and most profitable areas in New York City. The sensor data consists of taxi trip reports that include start and drop-off points, corresponding timestamps, and payment information. Data are reported at the end of the trip. Although the prediction tasks available in this application do not require real-time responses, it captures the data dissemination and query patterns of more complex upcoming transportation engines. An application that integrates additional data sources – bus, subway, car-for-hire (e.g., Uber), ride-sharing, traffic, and weather conditions – would exhibit the same structural topology and query rates that we use in our experiments while offering decision-making support in the scale of seconds. We implement the Urban sensing application [12] to aggregate pollution, dust, light, sound, temperature, and humidity data across seven cities to understand urban environmental changes in real-time. Since a practical deployment of environmental sensing can easily extend to thousands of such sensors per city, a temporal scaling of 1000× the native input rate can be used to simulate a larger deployment of 90,000 sensors.

Metrics. We focus on the performance metrics of query latency. Query latency is measured by sampling 5% of the tuples, assigning each tuple a unique ID and comparing timestamps at source and the same sink. To evaluate the scalability of DART, we measure how operators are distributed over nodes and how distributed schedulers are distributed over zones. To evaluate the adaptivity of DART, we cause bottlenecks by intentionally adding resource contention and we intentionally disable nodes through human intervention.

5.2 Query Latency

We measure the query latencies for running real-world IoT stream applications on the Raspberry Pis and VMs across a wide range of input rates.

Figure 7a and Figure 7b show the latency comparison of DART vs EdgeWise for (a) DAG queue waiting time and (b) DAG deployment time for an increasing number of concurrently running applications. We choose applications from a pool that contains dataflow topologies (DAGs) including `ExclamationTopology`, `JoinBoltExample`, `LambdaTopology`, `Prefix`, `SingleJoinExample`, `SlidingTupleTsTopology`, `SlidingWindowTopology` and `WordCountTopology`. EdgeWise is built on top of Storm. Both of them rely on a centralized master (Nimbus) to deploy the application’s DAGs, and then process them one by one on a first-come, first-served basis. Therefore, we can see that EdgeWise’s DAG queue waiting time and deployment time increase linearly as the number of applications increases. As such, the centralized master will easily become a scalability bottleneck. In contrast,

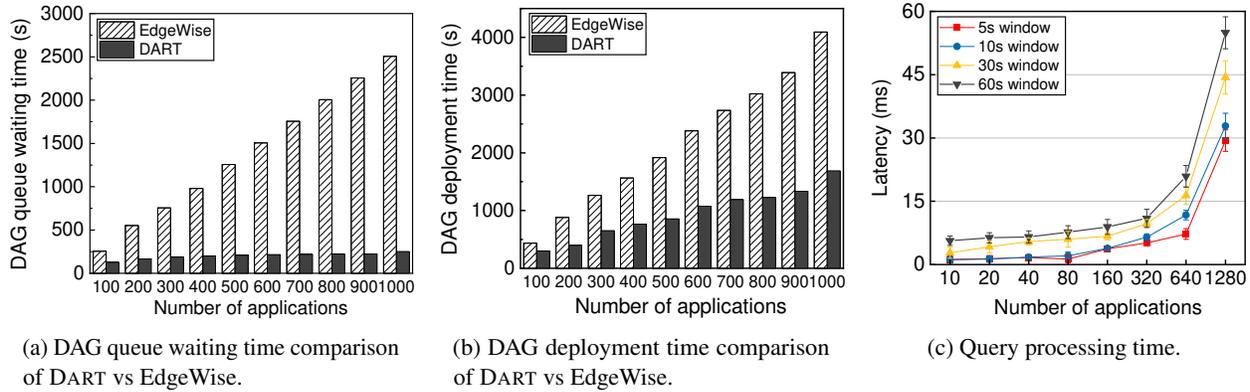


Figure 7: The latency comparison of DART vs EDGEWISE for (a) DAG queue waiting time, (b) DAG deployment time, and (c) query processing time by increasing the number of concurrently running applications.

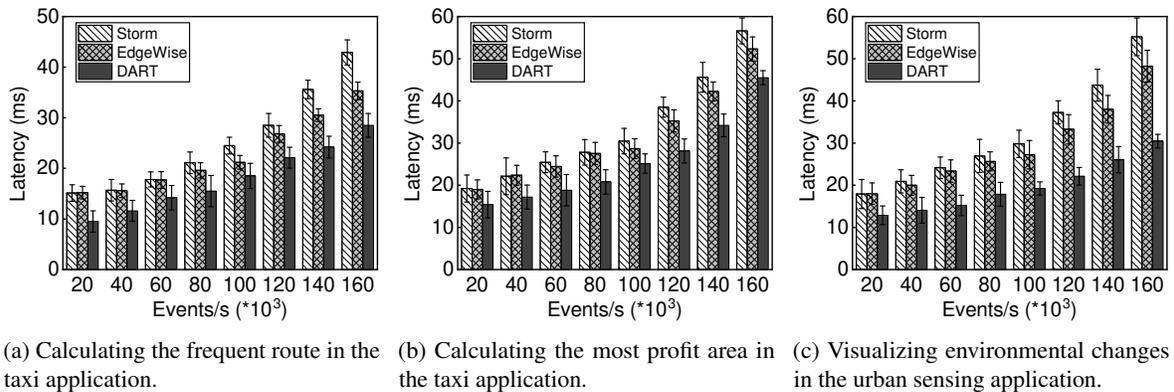


Figure 8: The latency comparison of DART vs Storm vs EdgeWise for (a) frequent route application, (b) profitable area application, and (c) urban sensing application.

DART avoids this scalability bottleneck because DART’s decentralized architecture does not rely on any centralized master to analyze the DAGs and deploy DAGs.

Figure 7c shows the query latency of DART for an increasing number of concurrently running applications. Results show that DART scales well with a large number of concurrently running applications. First, DART’s distributed schedulers can process these applications’ queries independently, thus precluding them from queuing on a single central scheduler which results in large queuing delay. This is similar to the idea that supermarkets add cashiers to reduce the waiting queues when there are many people in supermarkets. Second, DART’s P2P model ensures that every available node in the system can participate in the process of operator mapping, auto-scaling, and failure recovery, which could avoid the central bottleneck, balance the workload, and speed up the process.

The performance comparison results for running the frequent route application, the profitable areas application, and the urban sensing application are shown in Figure 8. In general, DART, Storm and EdgeWise [37] have similar performance when the system is under-utilized (with low input).

When the system is averagely utilized (with relatively high input), DART achieves around 16.7% ~ 52.7% less query latency compared to Storm. DART achieves 9.8 % ~ 45.6% less query latency compared to EdgeWise. This is because DART limits the number of hops between the source operators to sink operators within $\log(N)$ hops by using the DHT-based consistent ring overlay, and DART can dynamically scale operators when input rate changes. DART has better performance in the urban sensing application because this application needs to split data into different channels and aggregate data from these channels, which results in a lot of I/Os and data transfers that can benefit from DART’s dynamic dataflow abstraction. We expect further latency improvement under a limited bandwidth environment since DART selects the path with less traffic for the data flow by using the path planning algorithm.

5.3 Scalability Analysis

We now show scalability: DART decomposes the traditional centralized architecture of stream processing engines into a new decentralized architecture for operator mapping and query scheduling, which dramatically improves the scalability

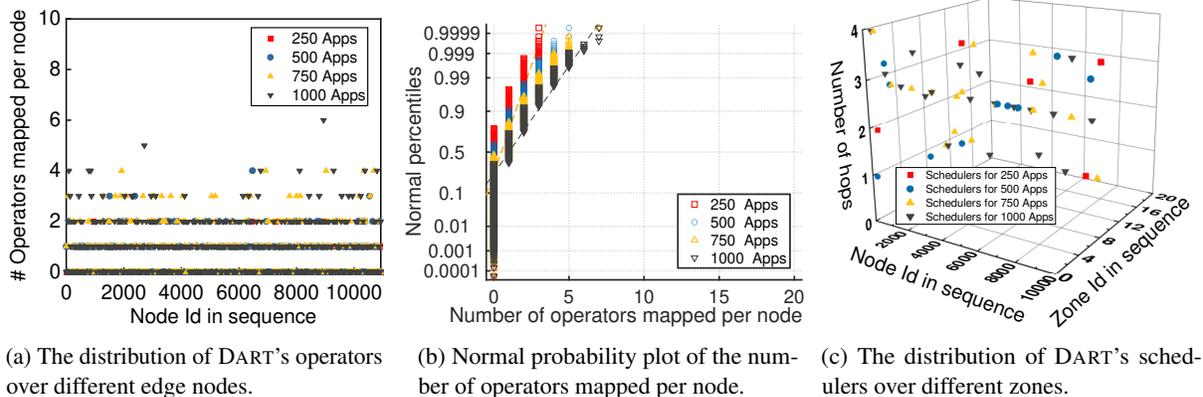


Figure 9: Scalability study of DART for the distribution of operators and schedulers over edge nodes.

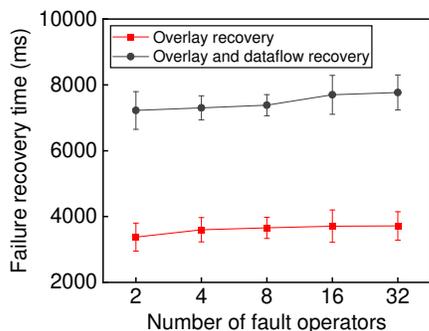


Figure 10: Overlay and dataflow recovery time.

for the system to scale with a large number of concurrently running applications, application's operators, and zones.

Figure 9a shows the mappings of DART's operators on edge nodes for 250, 500, 750, and 1,000 concurrently running applications, respectively. These applications run a mix of topologies with different numbers of operators (an average value of 10). Figure 9b shows the normal probability plot of the number of operators per node. Results show that when deploying 250 and 500 applications, around 96.52% nodes host less than 3 operators; and when deploying 750 and 1000 applications, around 99.84% nodes host less than 4 operators. From Figure 9a and Figure 9b, we can see that these applications' operators are evenly distributed on all edge nodes. This is because DART essentially leverages the DHT routing to map operators on edge nodes. Since the application's dataflow topologies are different, their routing paths and the rendezvous points will also be different, resulting in operators well balanced across all edge nodes.

Figure 9c shows the mappings of DART's distributed schedulers on edge nodes and zones for 250, 500, 750 and 1,000 concurrently running applications, and the average number of hops for these applications to look for a scheduler. For DART, it adds a scheduler for every new 50 applications. According to the P2P's gossip protocol, each application looks for a scheduler in the zone within $\lceil \log_{2^b} N \rceil$ hops, where $b = 4$. If

there is no scheduler in the zone or the number of applications in the zone exceeds a certain threshold, a peer node (usually with powerful computing resources) will be elected as a new scheduler. Results show that as the number of concurrently running applications increases, the number of schedulers over zones increases accordingly. All schedulers are evenly distributed over different zones. Most of the schedulers can be searched within 4 hops.

The above results demonstrate DART's load balance and scalability properties: (1) by using DHT-based consist ring overlay, the IoT stream application's workloads are well distributed over all edge nodes; and (2) DART can scale well with the number of zones and concurrently running applications.

5.4 Failure Recovery Analysis

We next show fault tolerance: in the case of stateless IoT applications, DART simply resumes the whole execution pipeline since there is no need for recovering state. In the case of stateful IoT applications, distributed states in operators are continuously checkpointed to the leaf set nodes in parallel and are reconstructed upon failures. We show that even when many nodes fail or leave the system, DART can achieve a relatively stable time to recover the overlay and dataflow topology.

Figure 10 shows the overlay recovery time and the dataflow topology recovery time for an increasing number of simultaneous operator failures. To cause simultaneous failures, we deliberately remove some working nodes from the overlay and evaluate the time for DART to recover. The time cost includes recomputing the routing table entries, re-planning the dataflow path, synchronizing operators, and resuming the computation. Results show that DART achieves a stable recovery time for an increasing number of simultaneous failures. This is because, in DART, each failed node can be quickly detected and recovered by its neighbors through heartbeat messages without having to talk to a central coordinator, so many simultaneous failures can be repaired in parallel.

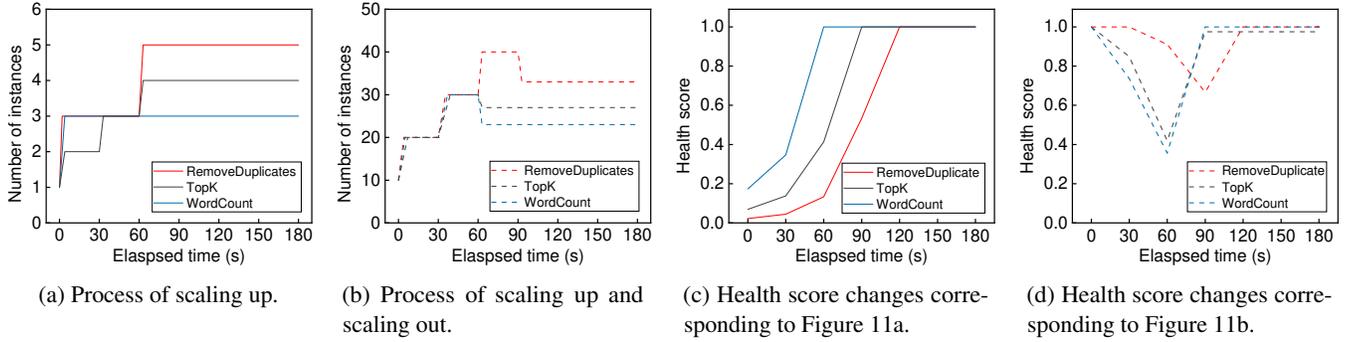


Figure 11: Adaptivity study of DART for the scaling up and the scaling out.

5.5 Elastic Scaling Analysis

Although scaling is a subject that has been studied for a long time, our innovation is that we use the DHT leaf set to select the best candidate nodes for scaling up or scaling out. Therefore, our approach does not need a central master to control, which is fully distributed. If many operators have bottlenecks at the same time, the system can adjust them all together. The periodical maintenance and update of the leaf set ensure that the leaf set nodes are good candidates, which are close to the bottleneck operator with abundant bandwidth, so there is no need for us to search for the appropriate nodes globally.

The auto-scaling process takes the system snapshot collected every 30 seconds for statistical analysis. We deploy three 4-stage topologies (RemoveDuplicates, TopK, WordCount). Figure 11a shows the process of scaling up only. The process starts from the moment of detecting the bottleneck to the moment that the system is stabilized. Figure 11b shows the process of scaling up and then scaling out. For this experiment, we put pressure on the system by gradually increasing the number of instances (tasks) (10 every 30 s) until a bandwidth bottleneck occurs (at 60 s for the blue line and the black line, and at 90 s for the red line). This bottleneck can only be resolved by scaling up. Results show that the system is stabilized by migrating the instance to another node.

Figure 11c shows how the health score changes corresponding to Figure 11a. Figure 11d shows how the health score changes corresponding to Figure 11b. Note that if the goal of pursuing a higher health score conflicts with the goal of improving throughput, we need to strike a balance between health score and system throughput by adjusting the health score function, i.e., aiming at a lower score.

5.6 Overhead Analysis

We evaluate the DART's runtime overhead in terms of the power usage and the CPU overhead. We run the same DEBS 2015 application [13] in Sec. 5.2 to calculate real-time indicators of most frequent routes in New York City with source rate at 100K *events/s*.

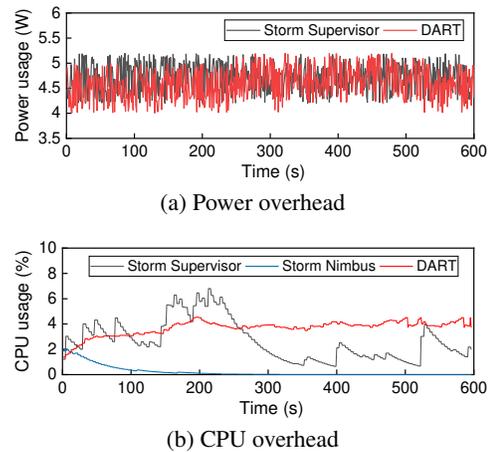


Figure 12: Overhead comparison of DART vs Storm.

Power usage. Most IoT devices rely on batteries or energy harvesters. Given that their energy budget is limited, we want to ensure that the performance gains achieved come with an acceptable cost in terms of power consumption. To evaluate DART's power usage, we use the MakerHawk USB Power Meter Tester [18] to measure the power usage of the Raspberry Pi 4. When plugged into a wall socket, the idle power usage is 3.35 Watt. Figure 12a shows the comparison of the averaged single device per-node power usage of DART node with Storm's supervisor when running the DEBS 2015 application. Results show that DART has less power usage with an average value of 5.24 Watt compared to Storm with an average value of 5.41 Watt, demonstrating that DART efficiently uses energy resources.

CPU overhead. Figure 12b shows the comparison of the CPU overhead of DART with Storm. Results show that DART uses more CPU than Storm Nimbus and Storm supervisor. DART continuously monitors the health status of all operators to make auto-scaling decisions to adapt to workload variations and bandwidth variations in the edge environment, while Storm ignores it. This CPU overhead is an acceptable trade-off for maintaining performance and could be further reduced with a larger auto-scaling interval.

6 Related Work

Existing studies can be divided into four categories: *cluster-based stream processing systems*, *cloud-based IoT data processing systems*, *edge-based data processing systems*, and *wide-area data analytics systems*.

Category 1: Cluster-based stream processing systems. Over the last decade, a bloom of industry stream processing systems has been developed including Flink [2], Samza [5], Spark [6], Storm [7], Millwheel [28], Heron [44], S4 [49]. These systems, however, are designed for low-latency intradatacenter settings that have powerful computing resources and stable high-bandwidth connectivity, making them unsuitable for edge stream processing. Moreover, they mostly inherit MapReduce’s “single master/many workers” architecture that relies on a monolithic scheduler for scheduling all tasks and handling failures and stragglers, suffering significant shortcomings due to the centralized bottleneck. SBONs [52] leverages distributed hash table (DHTs) for service placement. However, it does not support DAG parsing, task scheduling, data shuffling and elastic scaling, which are required for modern stream processing engines.

Category 2: Cloud-based IoT data processing systems. In such a model, most of the data is sent to the cloud for analysis. Today many computationally-intensive IoT applications [22, 23] leverage this model because cloud environments can offer unlimited computational resources. Such solutions, however, cannot be applied to time-critical IoT stream applications because: (1) they cause long delays and strain the backhaul network bandwidth; and (2) offloading sensitive data to third-party cloud providers may cause privacy issues.

Category 3: Edge-based data processing systems. In such a model, data processing is performed at the edge without connectivity to a cloud backend [19, 21, 58]. This requires installing a hub device at the edge to collect data from other IoT devices and perform data processing. These solutions, however, are limited by the computational capabilities of the hub service and cannot support distributed data-parallel processing across many devices and thus have limited throughput. It may also introduce a single point of failure once the hub device fails.

Category 4: Wide-area data analytics systems. Many Apache Spark-based systems (e.g., Flutter [39], Iridium [53], JetStream [55], SAGE [62], and many others [40, 41, 43, 65, 66]) are proposed for enabling geo-distributed stream processing in wide-area networks. They optimize the execution by intelligently assigning individual tasks to the best datacenters (e.g., more data locality) or moving data sets to the best datacenters (e.g., more bandwidth). However, they make certain assumptions based on some theoretical models which do not always hold in practice. For example, Flutter [39], Tetrium [40], Iridium [53], Clarinet [65], and Geode [66] formulate the task scheduling problem as a ILP problem. Pixida [43] formulates the task scheduling problem as a Min

k-Cut problem. They assume that the workload, the inter-DC transfer time, and the WAN bandwidth are known beforehand and do not change, which is rarely the case in practice. Moreover, these systems also suffer significant shortcomings due to the centralized bottleneck.

To our best knowledge, Edgent [1], EdgeWise [37], and Frontier [50] are the only other stream processing engines tailored for the edge. They all point out the criticality of edge stream processing, but no effective solutions were proposed towards scalable and adaptive edge stream processing. Edgent [1] is designed for data processing at individual IoT devices rather than full-fledged distributed stream processing. EdgeWise [37] develops a congestion-aware scheduler to reduce backpressure, but it can not scale well due to the centralized bottleneck. Frontier [50] develops replicated dataflow graphs for fault-tolerance, but it ignores the edge dynamics and heterogeneity.

7 Conclusion

Existing stream processing engines were designed for the cloud environments and may behave poorly in the edge context. In this paper, we present DART, a scalable and adaptive edge stream processing engine that enables fast stream processing for a large number of concurrent running IoT applications in the dynamic edge environment. DART leverages DHT-based P2P overlay networks to create a decentralized architecture and design a dynamic dataflow abstraction to automatically place, chain, scale, and recover stream operators, which significantly improves performance, scalability, and adaptivity for handling large IoT stream applications.

An interesting question for future work is how to optimize data shuffling services for edge stream processing engines like DART. Common operators such as `union` and `join` may require intermediate data to be transmitted over edge networks since their inputs are generated at different locations. Each shard of the shuffle data has to go through a long path of data serialization, disk I/O, edge networks, and data deserialization. Shuffle, if planned poorly, may delay the query processing. We plan to explore a customizable shuffle library that can customize the data shuffling path (e.g., ring shuffle, hierarchical tree shuffle, butterfly wrap shuffle) at runtime to optimize shuffling. We will release DART as open source, together with the data used to produce the results in this paper¹.

8 Acknowledgment

We would like to thank the anonymous reviewers and our shepherd, Dr. Amy Lynn Murphy, for their insightful suggestions and comments that improved this paper. This work is supported by the National Science Foundation (NSF-CAREER-1943071, NSF-SPX-1919126, NSF-SPX-1919181).

¹<https://github.com/fiu-elves/DART>

References

- [1] Apache Edgent - A Community for Accelerating Analytics at the Edge. <https://edgent.apache.org/>.
- [2] Apache Flink. <https://flink.apache.org/>.
- [3] Apache Flume. <http://flume.apache.org/>.
- [4] Apache Kafka. <https://kafka.apache.org/>.
- [5] Apache Samza. <http://samza.apache.org/>.
- [6] Apache Spark. <https://spark.apache.org/>.
- [7] Apache Storm. <http://storm.apache.org/>.
- [8] Apache Trident. <http://storm.apache.org/releases/current/Trident-tutorial.html>.
- [9] Apache ZooKeeper. <https://zookeeper.apache.org/>.
- [10] AVA: Automated Vehicles for All. <https://www.transportation.gov/policy-initiatives/automated-vehicles/10-texas-am-engineering-experiment-station>.
- [11] Cisco Kinetic Edge & Fog Processing Module (EFM). <https://www.cisco.com/c/dam/en/us/solutions/collateral/internet-of-things/kinetic-datasheet-efm.pdf>.
- [12] Data Canvas: Sense Your City. <https://grayarea.org/initiative/data-canvas-sense-your-city/>.
- [13] DEBS 2015 Grand Challenge: Taxi trips. <https://debs.org/grand-challenges/2015/>.
- [14] EdgeWise source code. <https://github.com/XinweiFu/EdgeWise-ATC-19>.
- [15] FAROO - Peer-to-peer Web Search: History. <http://faroo.com/>.
- [16] FreePastry. <https://www.freepastry.org/>.
- [17] Linux Traffic Control. <https://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html>.
- [18] MakerHawk USB Power Meter Tester. <https://www.makerhawk.com/products/>.
- [19] Microsoft Azure IoT Edge. <https://azure.microsoft.com/en-us/services/iot-edge>.
- [20] A new reality for oil & gas. https://www.cisco.com/c/dam/en_us/solutions/industries/energy/docs/OilGasDigitalTransformationWhitePaper.pdf, 2017.
- [21] Amazon AWS Greengrass. <https://aws.amazon.com/greengrass>, 2017.
- [22] Google Nest Cam. <https://nest.com/cameras>, 2017.
- [23] Netatmo. <https://www.netatmo.com>, 2017.
- [24] Soil Moisture Profiles and Temperature Data from SoilSCAPE Sites. https://daac.ornl.gov/LAND_VAL/guides/SoilSCAPE.html, 2017.
- [25] Autonomous cars will generate more than 300 tb of data per year. <https://www.tuxera.com/blog/autonomous-cars-300-tb-of-data-per-year/>, 2019.
- [26] HORTONWORKS: iot and predictive big data analytics for oil and gas. <https://hortonworks.com/solutions/oil-gas/>, 2019.
- [27] The ITS JPO's New Strategic Plan 2020-2025. <https://www.its.dot.gov/stratplan2020/index.htm>, 2020.
- [28] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, August 2013.
- [29] Mordecai Avriel. *Nonlinear programming: analysis and methods*. Courier Corporation, 2003.
- [30] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-scale Computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14*, pages 285–300, Berkeley, CA, USA, 2014. USENIX Association.
- [31] Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Randomized gossip algorithms. *IEEE/ACM Trans. Netw.*, 14(SI):2508–2530, June 2006.
- [32] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.
- [33] M. Castro, P. Druschel, A. Kermarrec, and A. I. T. Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):1489–1499, Oct 2002.

- [34] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM.
- [35] Federico Concone, Alessandra De Paola, Giuseppe Lo Re, and Marco Morana. Twitter analysis for real-time malware discovery. In *AEIT International Annual Conference, 2017*, pages 1–6. IEEE, 2017.
- [36] Avriilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: Self-regulating stream processing in heron. *Proc. VLDB Endow.*, 10(12):1825–1836, aug 2017.
- [37] Xinwei Fu, Talha Ghaffar, James C. Davis, and Dongyoon Lee. Edgewise: A better stream processing engine for the edge. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, 2019. USENIX Association.
- [38] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2351–2365, dec 2012.
- [39] Z Hu, B Li, and J Luo. Flutter: Scheduling tasks closer to data across geo-distributed datacenters. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, apr 2016.
- [40] Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. Wide-area analytics with multiple resources. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 12:1—12:16, New York, NY, USA, 2018. ACM.
- [41] Chien-Chun Hung, Leana Golubchik, and Minlan Yu. Scheduling Jobs Across Geo-distributed Datacenters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 111–124, New York, NY, USA, 2015. ACM.
- [42] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 783–798, Carlsbad, CA, 2018. USENIX Association.
- [43] Konstantinos Kloudas, Margarida Mamede, Nuno Preguiça, and Rodrigo Rodrigues. Pixida: Optimizing data parallel jobs in wide-area data analytics. *Proc. VLDB Endow.*, 9(2):72–83, oct 2015.
- [44] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.
- [45] Pinchao Liu, Hailu Xu, Dilma Da Silva, Qingyang Wang, Sarker Tanzir Ahmed, and Liting Hu. Fp4s: Fragment-based parallel state recovery for stateful stream applications. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020.
- [46] Petar Maymounkov and David Mazieres. Kademia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.
- [47] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas. Predicting taxi passenger demand using streaming data. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1393–1402, Sep. 2013.
- [48] Satoshi Nakamoto. Bitcoin : A Peer-to-Peer Electronic Cash System. Technical report, 2008.
- [49] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed Stream Computing Platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 170–177, Washington, DC, USA, 2010. IEEE Computer Society.
- [50] Dan O’Keeffe, Theodoros Salonidis, and Peter Pietzuch. Frontier: resilient edge processing for the internet of things. *Proceedings of the VLDB Endowment*, 11:1178–1191, 2018.
- [51] 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*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.
- [52] Peter Pietzuch, Jeffrey Shneidman, Jonathan Ledlie, Matt Welsh, Margo Seltzer, and Mema Roussopoulos. Evaluating dht-based service placement for stream-based overlays. In *International Workshop on Peer-to-Peer Systems*, pages 275–286. Springer, 2005.

- [53] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15*, pages 421–434, New York, NY, USA, 2015. ACM.
- [54] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. TimeStream: Reliable Stream Computation in the Cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys '13*, pages 1–14, New York, NY, USA, 2013. ACM.
- [55] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S Pai, and Michael J Freedman. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI'14*, pages 275–288, Berkeley, CA, USA, 2014. USENIX Association.
- [56] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.
- [57] Antony I T Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Middleware'01*, pages 329–350, London, UK, UK, 2001. Springer-Verlag.
- [58] Zhitao Shen, Vikram Kumaran, Michael J Franklin, Sathish Krishnamurthy, and Amit Bhat. CSA : Streaming Engine for Internet of Things. *Bulletin of the Technical Committee on Data Engineering*, 38(4):39–50, 2015.
- [59] W Shi, J Cao, Q Zhang, Y Li, and L Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, oct 2016.
- [60] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. Riotbench: An iot benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience*, 29(21):e4257, 2017.
- [61] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '01*, pages 149–160, New York, NY, USA, 2001. ACM.
- [62] R Tudoran, G Antoniu, and L Bouge. SAGE: Geo-Distributed Streaming Data Analysis in Clouds. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 2278–2281, may 2013.
- [63] 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*, pages 5:1—5:16, New York, NY, USA, 2013. ACM.
- [64] 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, SOSP '17*, pages 374–389, New York, NY, USA, 2017. ACM.
- [65] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. CLARINET: Wan-aware optimization for analytics queries. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 435–450, Savannah, GA, 2016. USENIX Association.
- [66] Ashish Vulimiri, Carlo Curino, P Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, NSDI'15*, pages 323–336, Berkeley, CA, USA, 2015. USENIX Association.

CrystalPerf: Learning to Characterize the Performance of Dataflow Computation through Code Analysis

Huangshi Tian¹, Minchen Yu^{1,2}, Wei Wang¹

¹HKUST, ²Huawei Technologies Ltd.

{htianaa,myuaj,weiwa}@cse.ust.hk

Abstract

Dataflow computation dominates the landscape of big data processing, in which a program is structured as a directed acyclic graph (DAG) of operations. As dataflow computation consumes extensive resources in clusters, making sense of its performance becomes critically important. This, however, can be difficult in practice due to the complexity of DAG execution. In this paper, we propose a new approach that *learns* to characterize the performance of dataflow computation based on *code analysis*. Unlike existing performance reasoning techniques, our approach requires *no code instrumentation* and *applies to a wide variety of dataflow frameworks*. Our key insight is that the source code of an operation contains learnable syntactic and semantic patterns that reveal how it uses resources. Our approach establishes a performance-resource model that, given a dataflow program, infers automatically how much time each operation has spent on each resource (e.g., CPU, network, disk) from past execution traces and the program source code, using machine learning techniques. We then use the model to predict the program runtime under varying resource configurations. We have implemented our solution as a CLI tool called CrystalPerf. Extensive evaluations in Spark, Flink, and TensorFlow show that CrystalPerf can predict job performance under configuration changes in multiple resources with high accuracy. Real-world case studies further demonstrate that CrystalPerf can accurately detect runtime bottlenecks of DAG jobs, simplifying performance debugging.

1 Introduction

Dataflow frameworks are deployed widely in the cloud for distributed machine learning (e.g., TensorFlow [2]), processing data streams (e.g., Flink [11]), and analyzing big data (e.g., Spark [81]). Dataflow frameworks provide a library of built-in operations (e.g., map, reduce, and tensor operators), which programmers use to compose a data processing pipeline, structured as a directed acyclic graph (DAG) of operations. Data flows between these operations and is thereby

transformed. The framework schedules DAG executions as parallel tasks on a cluster of machines.

However, dataflow computation routinely faces various performance issues such as resource contention, inefficient configuration, and poor scalability. Due to the complexity of DAG executions, troubleshooting performance issues in dataflow computation usually demands painstaking efforts even for skilled programmers [19, 82, 84].

A leading factor that makes performance debugging difficult is the lack of handy toolchains that can provide useful *performance-resource information* with actionable advices. Existing tools provided by popular frameworks often generate an overwhelming amount of low-level execution traces, which inexperienced developers may find difficult to even make sense of them. Many system solutions have hence been developed recently to simplify performance reasoning, but they either are designed for a specific framework (e.g., [31, 54, 55]) or require modifying the framework's source code with intrusive instrumentation (e.g., [34, 54]). As the codebases of dataflow frameworks are evolving rapidly, the instrumentation code itself needs to be updated frequently, mandating even more, repeated labor.

In this paper, we present CrystalPerf¹, a new performance characterization tool that requires *no code instrumentation* while *generally applicable* to an array of batch dataflow frameworks. Central to our design is a *learning-based performance-resource model* that, for a dataflow job (program), infers from the execution traces and the program source code how much time each operation has spent on different resources (e.g., CPU, network, disk). The model can then predict the program runtime under varying configurations or identify abnormal resource uses in execution.

CrystalPerf builds the performance-resource model for a dataflow job in two stages. In the first stage, it constructs a DAG execution profile (§4.2) from the log traces generated by the built-in profilers, where each node of the DAG repre-

¹Our tool serves as a *crystal sphere* where users can tell the performance of their dataflow jobs, either performance issues in the past execution or performance prediction in what-if scenarios.

sents an operation. For each operation, it extracts the execution details from the log traces, including the start and finish time, and the call trace of the executed functions. It then locates the source code of those functions in the framework’s codebase for further analysis.

In the second stage, CrystalPerf infers the resource use of each operation, characterized by the *resource-time* which measures the time proportion the operation spends on a resource type (e.g., 30% runtime on CPU, 70% on network). However, many frameworks provide no such information in the log traces, nor can it be obtained using common profiling tools. To tackle this challenge, our key insight is that *how an operation uses resources can be largely characterized by analyzing its source code.*² For example, an operation containing many routines for data compression (data transfer) is likely CPU-bound (network-bound). In fact, many dataflow frameworks implement operations using low-level functions provided by standard open-source toolkits and libraries (e.g., Netty [17], Akka [4], Parquet [18]), where experienced developers can clearly identify the bottleneck resources by just reading the source code of those functions. Our approach imitates this process with *neural network classifiers* that learn the resource use by analyzing the syntactic and semantic patterns of the function code (§3.2). The classifiers are trained using the source code of low-level functions (e.g., network and I/O primitives) collected from popular open-source projects with clear indications of resource uses. The trained classifiers can then be used in any framework and language.

Combining the DAG execution profile and the inferred resource-time, CrystalPerf can easily detect the performance bottleneck or abnormal resource uses. CrystalPerf can also predict the job performance in what-if scenarios (e.g., “how would the runtime change if the network is upgraded from 1Gbps to 10Gbps?”) by simulating the DAG execution under the assumed configuration (§5).

We have implemented CrystalPerf as a CLI tool (§6) with support of batch computations of Spark, TensorFlow, and limited cases of Flink [11]. To demonstrate its efficacy (§7), we use CrystalPerf to predict the runtime changes of dataflow jobs under varying configurations. Across six standard benchmarking workloads running in three frameworks, CrystalPerf predicts within an average deviation of 13.47%. Even compared with Monotasks [54], a meticulously architected Spark implementation for performance clarity, CrystalPerf makes runtime predictions with comparable accuracy. We highlight the framework generalizability and performance advantage of CrystalPerf over the existing dataflow characterization approaches in Figure 1. We further show through three real-world case studies that CrystalPerf can help troubleshoot various performance issues that are otherwise difficult to reason about for non-expert programmers.

²The source code includes both the program code *and* the associated documentation such as Javadoc.

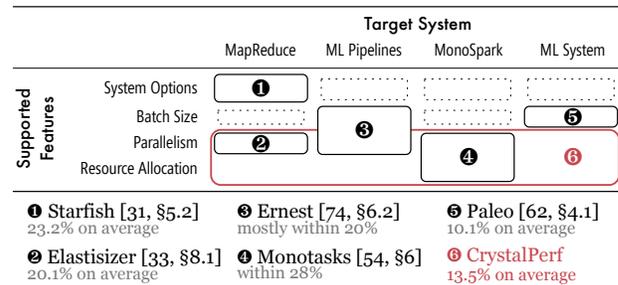


Figure 1: For performance reasoning, we compare CrystalPerf with existing works by answering “On which features can they predict the performance of what systems within how much deviation?”

2 Background and Motivation

In this section, we motivate the need of performance characterization for dataflow programs, where the key is to identify the performance-resource relationship (§2.1). We survey existing solutions and discuss their limitations, which are also the challenges we shall address (§2.2).

2.1 Performance Characterization

Performance characterization plays an indispensable role in the development of dataflow applications. Consider a programmer following the typical development workflow of coding, debugging, and optimizing. If the program runs much slower than expected, she may ask, “Is there any resource malfunctioning [28, 36]?” After fixing those issues, she may want to accelerate its execution, wondering, “What would happen if I put in use more resources [32, 38]?” If the performance remains unsatisfactory, she has to optimize the implementation, typically starting with the question, “What resource is the bottleneck during execution [34]?”

Understanding the performance of dataflow programs also allows cluster operators to better schedule DAG jobs and tune their configurations. State-of-the-art cluster schedulers need to predict how long each task of a DAG job would take if given a certain amount of resources [26, 27, 51, 71]; when deploying a user’s job, the operator also wants to tune the optimal configuration for fast job execution [5, 33].

The key to performance characterization is to identify the *performance-resource relationship*. For example, bottleneck detection and performance debugging can be done by knowing how much time the job spends on each resource; what-if questions, runtime estimation and configuration tuning requires predicting how long a job (or its tasks) would take under a different resource configuration.

However, characterizing the performance-resource relationship for dataflow programs is difficult. During execution, different operations may bottleneck on different resources, e.g., data compression being CPU-intensive, file writing

disk-intensive. For some operations like map in Spark, the actual bottleneck resource depends critically on the input (the map functions applied to an RDD). To make the problem even more complicated, some operations like join and shuffle require cross-machine synchronization and data communication. In practice, even skilled programmers cannot avoid painstakingly reasoning about and troubleshooting performance issues [19, 82, 84].

2.2 Related Work and Challenges

Existing profiling tools provided by popular dataflow frameworks produce an overwhelming amount of low-level traces, in which the relevant performance information is easily drowned out. Take TensorFlow as an example. When training ResNet50, the profiling traces of a single iteration include around 9,500 tensors, 3,000 operators, and 5,600 memory operations [1]. In fact, almost no built-in profilers provide high-level actionable advices for performance debugging. Many performance characterization tools have hence been developed recently for dataflow computation. However, they cannot fully tackle the challenges as summarized below.

Challenge 1: No explicit resource-time information. Though Linux (and other Unix-like systems) provides built-in utilities for time accounting and device monitoring, profiling the program’s resource-time (how long the program spent on each resource type) remains elusive. Linux’s I/O monitoring mechanism is mainly used for throughput tracing. For example, the per-process statistics given in /proc [49] only report the total amounts of read/written characters/bytes or sent/received bytes/packets. Without knowing the effective bandwidth and the data exchange size in execution, the actual I/O time cannot be estimated correctly. The counter-based observability tool perf and the industrial toolchains built on it [63] share the same problem.

The missing resource-time information cannot be uncovered by the recently proposed profiling tools either. Notably, SnailTrail [34] finds the critical path in distributed dataflow, but focuses on coarse-grained activities instead of fine-grained resource use information; Ernest [74] builds the performance model for advanced analytics like machine learning, but only includes input size and cluster scale as parameters. The lack of resource-time information limits their ability in addressing the resource-related issues.

Challenge 2: Heterogeneous hardware behavior. Heterogeneous hardware further complicates the performance model as dataflow operations usually have different resource patterns (e.g., data compression being CPU-intensive, file writing disk-intensive). Two approaches have been proposed in the literature. *Whitebox* approaches such as Starfish [31] build a detailed performance model for MapReduce execution, which cannot be applied to other frameworks. *Blackbox* approaches, such as CherryPick [5] and Ernest [74], con-



Figure 2: When a node (operation) is executing, attached is a sampling profiler for periodically sampling the call stacks. For each node, we build an *execution profile* to organize its resource-time information.

struct statistical machine learning models to correlate the resource allocations with the job performance, but it has to repeat the training process for each job type, making it time-consuming for ad-hoc or non-recurrent jobs.

Challenge 3: Cross-framework support. As dataflow computation prevails in a wide range of frameworks and application domains [2, 11, 12, 42, 47, 48, 79, 81], a general solution with cross-framework support is highly desirable. However, many existing works are tailored to a certain framework, e.g., AROMA [43], ARIA [75], Elastisizer [33] and Starfish [31] building specialized models for MapReduce programs, and DAC [80] designing an algorithm for tuning Spark jobs. Their designs are built on the internal details of the target frameworks and cannot be ported to other systems.

Our Answer In the coming sections, we incrementally develop our solution CrystalPerf to address the three challenges. We start to show in §3 that the time an operation spends on each type of resource can be learned from the source code and its call trace (addressing Challenge 1), using neural network classifiers which, once trained, can be applied to any framework and language (addressing Challenge 3). We then assemble the operation-level resource-time information into a job-level execution profile in §4. In §5, we further construct a performance-resource model to capture the program behaviors with respect to the allocated hardware resources such as CPU, GPU, memory, I/O and network (addressing Challenge 2). Our solution is novel as it requires no code instrumentation and applies to a wide array of dataflow frameworks.

3 Mining Resource-Time from Traces

In this section, we characterize the resource-time of an operation by exploiting the latent information in the call traces (§3.1). The periodic samples of call stacks can tell how long each operation (and its called functions) runs. If we could further obtain the resource use information of each function, then the resource-time of that operation becomes

available. We show that such information can be inferred from the function source code and documentation using neural network classifiers (§3.2–3.3). We validate the feasibility of this approach with a labeled and verified dataset (§3.4).

3.1 Obtaining Call Traces

Standard libraries provide built-in utilities to sample the call stacks, e.g., `backtrace` in GNU C Library and `AsyncGetCallTrace` in Java. CrystalPerf uses the sampling profilers [9] built on these utilities to collect the call traces of a dataflow job and its operation tasks periodically. Figure 2 shows a sample trace of a running `join` task in Spark.

In frameworks like TensorFlow and MXNet, CrystalPerf uses the built-in profiler as it records the start time and the duration of each operation. For JVM-based frameworks like Spark, Flink and Heron, CrystalPerf attaches a lightweight, non-intrusive sampling profiler to the JVM process, and aligns the trace with the time when an operation begins and finishes to extract the called functions. CrystalPerf then locates the implementation of those functions in the framework’s codebase for further analysis. Note that using similar sampling profilers for C# [25], JavaScript [68] and Go [23], the call traces can also be obtained in other frameworks.

3.2 Inferring Resource Use Patterns

With the durations of each operation and its called functions, we derive the resource-time information by analyzing how those functions use resources. As those functions are usually *low-level routines* (e.g., network and I/O primitives), they have clear resource use patterns which can be identified by a skilled programmer by simply examining the source code and its documentation. Following this intuition, we design two neural network classifiers to *infer* the resource patterns from the code and documentation, respectively. This approach requires no instrumentation and is not limited to a specific framework. Figure 3 illustrates how it works. We first embed the source code and documentation into vectors so that they can be processed by neural networks.

Code Embedding The key to code analysis is the *lexical and syntactic information*. The former includes the identifiers of variables, functions, classes and types; the latter refers to the abstract syntax tree (AST). CrystalPerf embeds them with a recursive autoencoder [67] (Figure 3). Following the embedding approach in [76], CrystalPerf parses the function code into an AST and augment it to a binary tree with some artificial nodes. CrystalPerf then constructs a vector representation for each AST node using the autoencoder. It starts by computing a word embedding for each identifier on a leaf node and recursively computes the embedding vector of a parent node from its children’s. Formally, let s_1 and s_2 be the two sibling nodes. The autoencoder computes their parents vector as $p = W[s_1 : s_2] + b$, where W and b are model

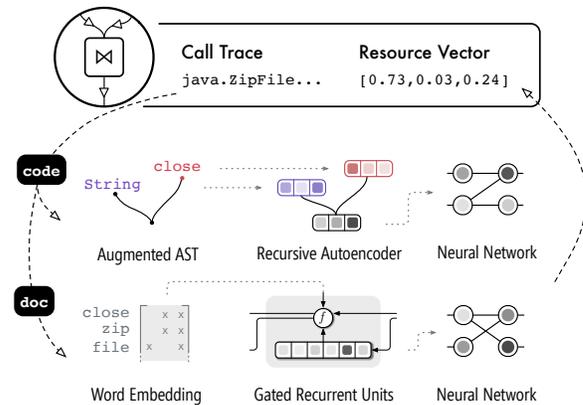


Figure 3: We design two classifiers that can infer the dominant resources for a function from the source code and documentation respectively.

parameters refined by training, and $[:]$ means concatenation. The vector of the root node gives the code embedding.

Documentation Embedding A function may have an associated documentation explaining its performed task and usage, written in natural language. To embed a function documentation, CrystalPerf removes the contained punctuations and hyperlinks and encodes the words into vectors with Flair [3], a language model with state-of-the-art performance. CrystalPerf then encodes the whole document with gated recurrent unit (GRU) [14] which can take an arbitrarily long input sequence. GRU has internal states and works in a recursive manner: it takes an input element, updates state, produces output and repeats. CrystalPerf sequentially feeds the word embeddings to GRU and takes the state from the last iteration as the document embedding.

Inference Procedure With the code and documentation embeddings calculated, CrystalPerf infers the function’s resource uses with two pre-trained neural network models (see §3.3). Both models output a *resource vector*, where each element is the probability of the function using a certain type of resource (e.g., 0.7 in CPU and 0.3 in network), given by the code or documentation analysis. CrystalPerf takes the average of the two predictions as the inferred resource vector³. Our design currently supports three resource types and the output is in the form of $\langle \text{CPU}, \text{Disk}, \text{Network} \rangle$. We next describe the neural network design and training.

3.3 Model Design and Training

Critical to our approach are the two neural network models that learn the resource vector from the function code and documentation. This problem resembles document classifi-

³§B in the supplemental material [70] provides the justification for averaging.

Table 1: The prediction quality of the classifier on the collected dataset. The higher-than-random accuracies imply the learnability of resource information from the source code.

	Accuracy	F1 Score
CPU	77.12% ± 7.39%	0.869 ± 0.046
Disk	82.46% ± 5.63%	0.903 ± 0.034
Network	82.90% ± 5.26%	0.906 ± 0.032
Average	80.83% ± 4.60%	0.893 ± 0.028

cation [78,83], in which a neural network model is trained to predict the probability of a document belonging to a certain class. In our design, we adopt the single-layer neural network (i.e., linear model) because it supports variable lengths in both input and output, allowing us to adjust the dimension of embeddings and incorporate new resources.

To train the two classification models, we collect source code from open-source projects with clean designs and comprehensive documentation. In each project, we choose the source files that have apparent resource uses, such as TCP and UDP services (network-bound), compression and hashing (CPU-bound), file operations (I/O-bound). We extract the selected functions together with their documentation and label their resource patterns. In total, we have collected around 5,000 low-level functions from five popular open-source libraries, including FS2 [22], Twitter Util [72], Play Framework [59], Scalaz [65] and Scala Standard Library.⁴ It took two PhD students 5 hours to manually label those functions.⁵ The labeled resource vectors are in the form of <CPU, Disk, Network>, with the dominant resource set to 1 and the others 0 (one-hot vector). Although some functions use more than one resource, only labeling the dominant one is still feasible as our training dataset only includes low-level libraries where most functions rely on just one type of resource. Admittedly, such labeling inevitably involves subjectiveness and imprecision, but our sensitivity analysis (§7.4) shows that CrystalPerf is robust against the labeling errors.

We have implemented the two classifiers with PyTorch [57].⁶ The classification models, once trained, are *not restricted* to certain frameworks or programming languages. First, our model design accepts arbitrary words as its input because the word embedding techniques, such as Flair [3], work on the level of characters and allow unseen words beyond the training corpus. Second, regardless of the frameworks, their documentation is all written in natural language,

⁴The collected dataset are released at [69].

⁵Manual labeling is feasible as long as the project follows modern design guidelines (e.g., modularity, high cohesion, low coupling). For instance, in a well modularized project, functionally similar codes tend to appear in the same module, so it is feasible to “batch process” them.

⁶The documentation classifier is optimized with stochastic gradient descent (SGD). Its learning rate is set to 0.1 and annealing rate 0.5. For the more complicated code classifier, we choose a more stable optimization algorithm, Limited-memory BFGS, with its step size as 1×10^{-6} .

Table 2: Top 5 words that make the classifier predict a certain category. Most of them appear in *both code and documentation*, except that the underlined ones only in code.

Category	Words
CPU	<u>engine</u> , entry, stream, key, <u>certificate</u>
Disk	file, error, tar, info, name
Network	socket, sock, <u>send</u> , result, address

enabling the reuse of the documentation classifier. Third, many keywords and syntaxes (e.g., condition and repetition) are commonly used in different programming languages, a common ground for our code classifier to exploit. Our evaluation confirms the generalizability of our design: the models trained with Scala code can make accurate predictions in frameworks written in Java and C++ (§7.1).

3.4 Model Validation

We design a model validation test to verify that both the code and documentation contain useful resource information that can be learned correctly by our classifiers. We collect a set of functions from Python standard library and OpenJDK, *manually verify* their resource uses with `strace`, train our classifiers over them, and compute the prediction accuracy. As shown in Table 1, the trained models can identify the dominant resource of a code snippet with accuracy over 77%. The F1 scores⁷ are higher than 0.86, suggesting no possible skewness towards any classes. These results show that our classifiers can accurately identify resource uses from the source code. We believe this also holds true for other codebases as we randomly select the source files from standard libraries.

We next inspect what features learned by our classifiers make major contributions to the prediction results. We turn to LIME [64], a popular framework that can explain what words are the key to making a certain prediction. Table 2 lists the most frequent words that appear in the explanations. As one may expect, the computation of “key” or “certificate” is CPU-intensive and “file” operations involve disk I/O. They indicate that our models have indeed learned meaningful patterns for resource inference. We further compare the predictions made separately by the code and documentation classifiers, and have two major findings: (1) the code and the documentation classifiers have roughly the same level of prediction ability, and (2) they make more accurate predictions on *longer* and *more diverse* code and documentation. We defer the detailed analysis to supplemental materials (§A).

Now that we have the function-level resource information,

⁷F1 score is the harmonic mean of the precision and recall, whose value is within 0 and 1. The larger the value is, the more accurate and the less biased the classifier is.

it remains a challenge to build a job-level profile on top of it, which we address in the next section.

4 Profiling Resource Usage of Job

The previous section infers resource-time for a single function, yet a performance-resource model requires the information about each operation (which may call multiple functions) and the whole job. In this section, we piece together the function resource-time into a resource vector for each operation (§4.1), which unifies the varied resource patterns and captures the time proportion spent on each resource. We then assemble the operation information into a job-level DAG execution profile (§4.2), which forms the basis of performance debugging and prediction.

4.1 Resource Vector

We characterize the resource uses of an operation with a *resource vector*, where each component is the *proportion* of runtime it spent on one resource during execution. For example, the resource vector $\vec{R} = (p_{\text{cpu}}, p_{\text{disk}}, p_{\text{net}}) = (0.15, 0.03, 0.82)$ indicates that the operation mainly uses CPU, disk, and network, with each accounting for 15%, 3% and 82% of the operation execution time. We define resource vectors with *relative* time proportions instead of the absolute scale because the former can be more conveniently handled in machine learning. As the proportions add up to one, our definition implicitly assumes *sequential resource use* during execution. Though this may not be exactly the case (e.g., pipelining in Spark), it serves as a good approximation in many frameworks because operations are typically low-level functions performing simple tasks (e.g., math operation in TensorFlow, primitive operation in Flink). Furthermore, for those operations spending the vast majority of time on one resource, which is usually the case, neglecting concurrent uses on the other resources has a negligible impact to runtime prediction. One piece of empirical evidence is *Monotasks* [54], which rearchitects Spark operators to avoid resource pipelining and merely incurs 9% increase in runtime. Our evaluation also confirms the validity of the sequential resource use model (§7.1).

CrystalPerf infers the resource vector of each operation from its call traces. A stack typically has some system function in the bottom (e.g. `Thread.run`) and the most recently called function at the top. Since the topmost function can be some library or native routine, we scan downwards until we find a function whose source code is available (i.e., some function in the dataflow framework). We then apply the classifiers introduced in §3.2 to infer its resource use, and take the output as its resource vector. As the profiler samples at fixed intervals, we simply assume that the functions in a given stack have remained execution in that interval. We take the average of all the inferred vectors and use it as the

resource vector of the operation in that interval.

4.2 DAG-Based Execution Profile

With the resource vector of operations, we establish the execution profile for a dataflow job in two stages. In the first stage, we organize a DAG to represent the job execution, with necessary modification for certain frameworks (details following next). In the second stage, we annotate each node (operation) with its execution details and the resource-time inferred from its call traces.

Dataflow frameworks expose DAG information through monitoring APIs or runtime logs. In most dataflow frameworks, the job DAG provided in the log trace (e.g., TensorFlow computation graphs and Flink dataflows) can be directly used to represent the internal computation structure, i.e., the operations and their dependencies. Yet a special treatment is needed for Spark, in which a job is decomposed into multiple stages, each containing multiple parallel tasks [81]. We therefore turn to a *hierarchical* DAG representation where a node itself can be a smaller DAG. After constructing the DAG, we associate the profiled execution details to each node, including the start and end time of the operation, the functions it called during execution, the source code of those functions, and the inferred resource vector of the operation. Figure 2 shows a sample profile attached to a join operation in Spark.

The DAG execution profile forms the basis for debugging performance issues. To identify the execution bottleneck, CrystalPerf follows the operation execution on the critical path and sums up the time spent on different resources. The one that takes the longest time is identified as the *bottleneck resource*. The execution profile can also be used to troubleshoot configuration problems by displaying the resource-time breakdown of each operation. For example, in one of our case studies in §7.5, a machine learning job has spent a significant portion of time on I/O. Even though I/O is not the major bottleneck compared with computation, its overuse still suggests a misconfiguration problem for the training job.

5 Performance-Resource Model

The DAG execution profile established in the previous section can be used to characterize the job behaviors in the past execution. In this section, we take a step forward by predicting how the job would perform in what-if scenarios with different resource configurations. We categorize hardware resources into three classes following the von Neumann architecture and model their impacts to the job performance respectively.

Computing Devices such as CPU, GPU or other emerging processors [21, 39] can affect job performance in two ways. (1) Changing the number of devices (e.g., CPU cores) may

Algorithm 1 Predict job runtime under target computing devices.

– N, N' : original and target number of computing devices
– s, s' : original and target computing speed

- 1: **function** PREDICTRUNTIME(s, N, s', N')
- 2: Replay tasks in DAG on N' devices.
- 3: $w \leftarrow$ slowdown of the first-wave tasks due to warm-up
- 4: Scale first-wave task runtime by w \triangleright warm-up effect
- 5: **for** each task in replayed DAG **do**
- 6: $v_{\text{cpu}} \leftarrow$ CPU component of resource vector
- 7: $t_{\text{cpu}} \leftarrow v_{\text{cpu}} \times$ task runtime
- 8: Scale t_{cpu} by s/s'
- 9: Traverse the DAG and compute the updated job runtime

change the degree of parallelism of tasks. (2) Utilizing devices with different processing speeds may change the task runtime. Taking both factors into account, Algorithm 1 estimates the job runtime under a parallelism change from N to N' and a speed change from s to s' . The first part (Line 2-4) reschedules the parallel tasks over N' devices and *simulates* their executions to estimate the runtime.⁸ During the simulation, we distinguish the sequential execution from the parallel, following the insight of Amdahl’s law [6]. As a concrete example, some Spark tasks are executed in a single thread despite the existence of multiple cores, so we keep it sequential in the simulation. We also consider the *warm-up effect*. That is, the first wave of tasks take longer time to complete than the rest, due to various forms of cache (e.g., the instruction cache of CPU, the data cache in distributed storage systems [10, 24] the code cache in JVM [46], etc.). Therefore, if a non-first-wave task is rescheduled to the beginning, we adjust it by a warm-up factor w and vice versa. After the adjustment, the second part (Line 5-8) of the algorithm rescales the compute time of each task by s/s' as CPU time is inversely proportional to the speed [30].⁹

Memory plays a passive role in job execution. Once a job is allocated sufficient memory, more allocation will not make it run faster. Conversely, when a job runs short of memory, frequent garbage collections or paging prolongs its execution. We characterize the relationship between memory and runtime with a *reversed roofline model* [77]—a linearly decreasing function followed by a constant lower limit, where the turning point is chosen by summing up the data size used in the program (e.g., RDD blocks in Spark, tensors in TensorFlow, network buffers in Flink) and the slope is the ratio between the I/O speed of in-memory and on-disk data access. The intuition behind is that the memory is sufficient as long as it can accommodate all data and the delay is primarily

⁸Currently, the replay simulation does not consider the task scheduling but follows the original task order. It is feasible for users to provide a customized scheduler as a plugin (§6).

⁹Approximating speed as a constant follows the precedents in Paleo [62] and Starfish [31].

Algorithm 2 Predict job runtime under target I/O devices.

– B, B' : original and target I/O bandwidth
– S_B : buffer size
– o : overhead of processing buffer for one time

- 1: **function** PREDICTRUNTIME(B, B')
- 2: **for** each task in job DAG **do**
- 3: $v_{\text{I/O}} \leftarrow$ I/O component of resource vector
- 4: $t_{\text{I/O}} \leftarrow v_{\text{I/O}} \times$ task runtime
- 5: $D \leftarrow t_{\text{I/O}} \cdot B$ \triangleright estimated data size
- 6: $t_p \leftarrow (D/S_B) \cdot o$ \triangleright processing overhead
- 7: $t_c \leftarrow t_{\text{I/O}} - t_p$ \triangleright communication time
- 8: Scale t_c by B/B' .
- 9: Traverse the DAG and compute the updated job runtime.

caused by data transfer between memory and disk. We admit that other factors, such as garbage collection, may also affect the runtime, but our evaluation shows that such a simplified model is sufficient to predict the runtime accurately under a wide range of memory changes (§7). We leave it as a future work to model more complicated memory effects on dataflow jobs.

I/O and Network Devices We adopt a buffered I/O model [56] where the data is first placed inside a buffer and then transferred. While other I/O variants do exist, the buffered I/O model remains the most common practice used in Linux API [50] and Java Standard Library [37]. In Algorithm 2, we divide the I/O time into the *buffer processing time* and the *data communication time*. We assume that data are sequentially placed into a buffer and processing a buffer incurs a fixed overhead. The total processing overhead is hence estimated as the number of buffers times the overhead (Line 6). As for the parameters, we set the overhead o the same as the measurement result reported in [58], and buffer size S_B the default values in Linux. The communication time is then rescaled by B/B' to simulate the change in bandwidth.¹⁰

Putting It All Together CrystalPerf combines the execution profile with resource-runtime model to characterize job performance. Figure 4 illustrates the main use cases. For debugging, CrystalPerf generates a detailed report of the time spent on each resource to help users understand the potential issues of the program. For prediction, we consider the scenario where a user profiles a job under a given configuration and wonders how the job completion time would change with a different configuration. CrystalPerf makes the prediction by constructing the DAG execution profile and computing the operation runtime in the new configuration following the aforementioned device models. In both use cases, CrystalPerf requires no code instrumentation nor restricts to a certain framework.

¹⁰Following prior works [31, 62], we assume constant bandwidth in our modelling.

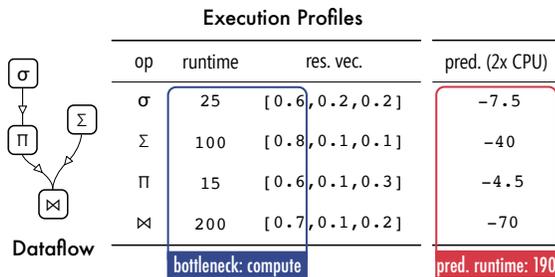


Figure 4: Given an executed job (left), CrystalPerf can either ① analyze resource usage or ② predict its job runtime under another resource configuration (right).

6 Implementation

We have implemented CrystalPerf as a CLI tool in around 4,000 lines of Python and Scala code, which is open-sourced at [69]. Our implementation currently supports three frameworks, Spark, Flink, and TensorFlow.

Pluggable Architecture As a framework-independent tool, CrystalPerf sets to provide a general support for various frameworks, even the future ones. So it has to accommodate different logging conventions, trace formats, DAG representations, etc. We provide plugin mechanisms wherever possible to maximize the usability and generalizability. Our implementation consists of three modules for parsing runtime logs and traces, annotating DAG with resource vectors, and predicting performance based on the given models. The parser accepts plugins that can update with the changing log formats. The annotator supports plugins for locating source code with different directory structure or naming convention. The predictor allows plugins for customized performance models, such as Starfish [31] and Paleo [62]. In our implementation, the supported frameworks (i.e., Spark, Flink and TensorFlow) share the same base modules but all have their specific plugins.

Usage Instructions To use CrystalPerf, users need to enable logging and profiling when executing dataflow jobs. For most JVM-based systems, this can be done by just adding a configuration line and attaching a jar package of an off-the-self profiler.¹¹ Users simply specify the location of the logs and traces in the CLI, and let CrystalPerf extract the runtime information and construct the DAG profile automatically. CrystalPerf provides two modes. (1) For debugging, CrystalPerf outputs the time spent on each resource. (2) To predict the performance under a certain resource configuration, users have to pass that configuration to the CLI, together with the original one. CrystalPerf applies the performance models and outputs the predicted runtime.

¹¹For frameworks such as TensorFlow and MXNet, the log traces already contain the operator names, and there is no need to specify a profiler.

7 Evaluation

In this section, we evaluate CrystalPerf in various scenarios. The highlights of our evaluation are summarized as follows:

- CrystalPerf can predict the job runtime of three frameworks under multiple types of resource variations within an average deviation of 13.47% (§7.1).
- CrystalPerf can predict the performance of a Spark benchmark with accuracy comparable to Monotasks [54], the heavily intrusive state-of-the-art operating on a version of Spark that simplifies performance reasoning (§7.2).
- The neural network classifiers are resistant to the inaccuracy in the labels of training dataset and generalizable to different frameworks (§7.4).
- CrystalPerf can effectively help users identify bottleneck resources and address performance issues (§7.5).

7.1 Performance Prediction

Methodology We evaluate CrystalPerf against six workloads in Spark, Flink, and TensorFlow with different resource variations, as summarized in Table 3. For each workload, we first run it under a baseline configuration and measure its runtime. We then change the resource configuration and use CrystalPerf to predict the new runtime. To measure the accuracy, we rerun the job on a real cluster with the same configuration as in prediction. We report the predicted and the actual runtime in the experimental results. We use *deviation* as a metric to measure accuracy, defined as $|p-a|/a$, where a is the actual runtime and p the prediction.

Spark Figure 5 shows the runtime predictions given by CrystalPerf for two TPC-H queries (see Table 3) under various configurations, where the second bar group in the central graph shows the baseline execution. The average deviation of the predictions is $13.49\% \pm 8.57\%$, demonstrating the effectiveness of CrystalPerf for Spark applications. Note that such deviations are evenly distributed around the actual runtime, an indication that our approach shows no systematic over- or underestimation but mainly suffers from random errors.

Flink For stream processing, we choose two widely used benchmarks [34, 73]. As CrystalPerf does not natively support long-running jobs, we define the *latency* of a streaming application as its runtime because it measures the time of a data batch (i.e., a data buffer in Flink) traversing through the DAG. Therefore, throughout the experimentation of Flink, we run the application for 10 minutes and report the average latency as its runtime. The resource variation includes the change of CPU share and network bandwidth, where CPU share determines the fraction of CPU time granted to a given program.

Figure 6 presents the prediction results of Flink applications, where the average deviation is $12.70\% \pm 10.11\%$.

Table 3: Summary of the workloads used in performance prediction (§7.1).

Framework	Workload	# Instances	Instance Type	Varied Resources
Spark v2.4.3	TPC-H [60] query 6 (short) and 9 (long) with scale factor 100	16	AWS m5.xlarge (4 cores, 16GB mem, 10Gbps)	# cores, memory, network bandwidth
Flink v1.7.2	Yahoo Streaming Benchmark [13] and Dhalion Benchmark [20]	11 (1 master and 10 workers)	AWS m5.xlarge	CPU share, network bandwidth
TensorFlow v1.13	ResNet [29] and VGG [66] on a flower image dataset	8 (1 master, 4 workers and 3 servers)	GCP n1-16-standard (16 CPUs, 60GB mem, Tesla P100 GPU)	computing devices, # cores, memory

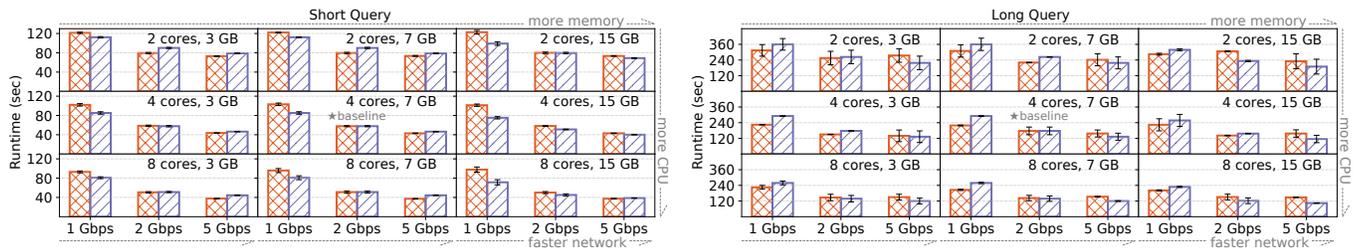


Figure 5: Given the baseline execution (located at the center) of two TPC-H queries, CrystalPerf predicts their runtime under the rest 26 resource configurations. Each bar pair shows one prediction and its corresponding actual runtime (Prediction, Actual).

Again, the deviations are evenly distributed instead of biasing towards over- or underestimation. Moreover, as our classifiers are trained with Scala code (§3.3) whereas Flink is mainly written in Java, the results demonstrate the generalizability of the classification models to other programming languages. Such desirable property results from the fact that programmers tend to use meaningful names [35] and hence those resource-related terminologies are used in different languages.

TensorFlow As for distributed machine learning, we use a managed version of TensorFlow (v1.13) provided by AI Platform [15] in Google Cloud. The training programs adopt the parameter server architecture [45] with asynchronous parallel (ASP) scheme. Because the runtime characteristics of parallel machine learning are highly repetitive across iterations, we report the *average iteration time* as the job completion time. As the iterations in ASP may not be temporally aligned, we take the average of each worker’s iteration time. The resource variation in learning applications include upgrading CPU to GPU, using a different number of virtual CPUs and changing the memory size. Figure 7 shows the prediction results given by CrystalPerf. The average deviation is $14.22\% \pm 11.77\%$. This again confirms the satisfactory prediction accuracy of our approach and the generalizability of our classification models.

7.2 Comparison with Monotasks

As summarized in Figure 1, Monotasks [54] is the most closely related to our work, so we make a comparison to it by predicting the performance of Big Data Benchmark (BDB) [7]. Note that Monotasks has devised its own variant of Spark for performance clarity called *MonoSpark*, which decomposes the pipelined usage of multiple resources so that each compute task only uses one resource. Following the description in §6.2 in [54], we reproduce the experiment in the same settings: first run the BDB workloads on MonoSpark with two disks and let CrystalPerf predict the runtime under the configuration of one disk. As most of MonoSpark’s implementation relies on Spark (including logging), CrystalPerf can reuse its Spark interface.

Figure 8 plots the actual and the predicted runtimes of all queries in the benchmark. CrystalPerf achieves an average deviation of $12.53\% \pm 5.10\%$, while Monotasks reports most errors within 9% and an exception as high as 28% (cf. Figure 12 in [54]). It is worth mentioning that Monotasks uses a whitebox performance model that is specifically tailored to the decomposed design of MonoSpark. In comparison, CrystalPerf requires no such tailoring but still can capture the resource pattern of MonoSpark. This once again confirms that (1) CrystalPerf has comparable prediction ability with the intrusive state of the art, and (2) its effectiveness is not restricted to a particular framework.

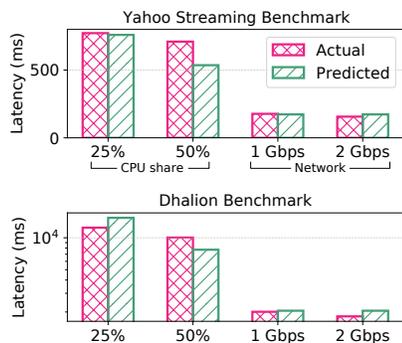


Figure 6: Performance prediction of Flink workloads. The baseline machine uses 100% CPU share and 5Gbps network.

7.3 Microbenchmarking Performance Models

Having shown the predicting ability of CrystalPerf, we need to confirm that the constructed performance models match the real execution. We scrutinize the Dhalion Benchmark running with 100% CPU share (baseline) and 25% (prediction). We choose this case because Flink does not have complicated pipelining so that we could easily check the real resource usage based on the profiling traces. It is worth mentioning that the traces also include the warm-up period (i.e., Java JIT compilation), so we exclude them from the traces and only take samples after the system enters a stable state. The benchmark mainly has two operators, FlatMap and Window. From the latency measurements, we find that they respectively incur 1583.63ms and 2322.50ms latency in the 100% case, and 2961.75ms and 7181.63ms in 25%.

Resource Vector We first verify the CPU entries in the resource vectors of both operators, because they are the key components in prediction. CrystalPerf infers that FlatMap and Window respectively spend 15.13% and 85.05% of the operation time on computation, with an average percentage of 56.70%.¹² We manually examine the profiling traces of the baseline case and find the most dominant computation functions are StreamFlatMap.processElement and WindowOperator.processElement. They appear in 65.46% of the call stack samples, which is close to the inferred percentage of CPU usage.

Performance Model Following our resource-time models (§5), CrystalPerf predicts that FlatMap and Window will take 2296.26ms and 8244.88ms, respectively, if running with 25% CPU share. Again, they are within a tolerable error range when compared with the measured durations. Furthermore, we observe that the occurrences of NioEventLoop.run, a network-related function, increases

¹²It is computed as a weighted average: $(15.13\% \times 1583.63 + 85.05\% \times 2322.50) \div (1583.63 + 2322.50) = 56.70\%$

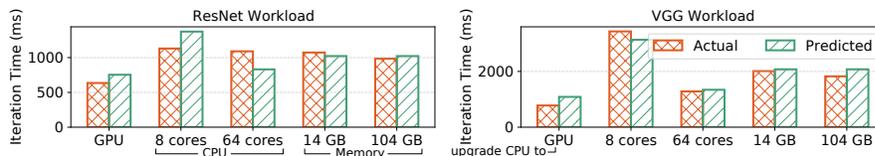


Figure 7: Performance prediction of TensorFlow workloads. The baseline machine uses 16 virtual CPU cores and 60GB memory.

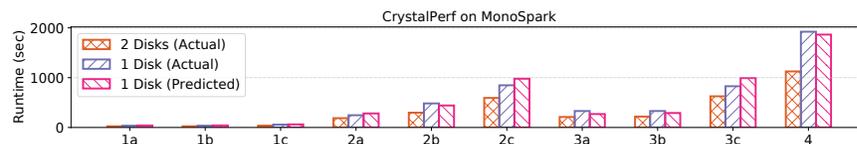


Figure 8: Performance prediction of Big Data Benchmark (BDB) on MonoSpark.

from 3.20% of the samples to 4.10%. This shows how the slowdown of CPU could prolong the processing time of network I/O, confirming our intuition used in modeling resource changes.

7.4 Classifier Sensitivity and Generalizability

Sensitivity to Labeling Quality Recall in §6 that the resource vectors of the low-level functions we collected in the training dataset are manually labeled, and it is difficult to check their accuracy due to the lack of the ground truth. Henceforth, we instead evaluate the robustness of CrystalPerf with respect to the labeling quality. We reuse the training dataset and perturb the labels of 10%/20% data samples. For instance, for the data samples originally labeled as CPU-dominant, we change 1.67% of them to disk and another 1.67% to network, and repeat the process for other two resources to reach the 10% fraction. As shown in Figure 9, the resource classifiers used in CrystalPerf are robust against the labeling errors.

Generalizability Apart from robustness, generalizability is another desirable property because the users don't want to retrain the classifiers for each single framework. Although §7.1 confirms that CrystalPerf can work on different systems with the same classifiers, here we zoom in on the predictions they make on various programming languages. For the test set, we collect 105 low-level functions from Spark (Scala), 194 from Flink (Java), 302 from TensorFlow (Python), and label them in the same way as in §3.3. The prediction results shown in Table 4 indicates that our classifiers, trained on low-level Scala code, predict similarly well across three frameworks, which partially explains the generalizability of CrystalPerf. One noteworthy detail is that sometimes TensorFlow has even higher accuracy than Spark. This is because the tested TensorFlow functions include some example code where they are lengthily documented and thus appear more

Table 4: The classifiers trained with Scala code perform similarly on the chosen systems, implying the generalizability of our approach. (Accuracy/F1 Score)

	Code Cls.	Doc. Cls.
Spark	81.0%/1.797	76.5%/1.673
Flink	73.3%/1.664	74.8%/1.589
TensorFlow	79.9%/1.752	77.8%/1.652

informative to the classifier model.

7.5 CrystalPerf in Action: Case Studies

We further conduct three real-world case studies to demonstrate how CrystalPerf could help users identify resource bottleneck and address performance issues with ease.

Diagnosing Slow Operation A user implemented a Spark program to process a dataset stored in HDFS [8]. However, the program took much longer time than she had expected. We investigated the case by reproducing it with the same configuration as original. We launched a six-node cluster on AWS EC2 with m5.4xlarge instances (16 cores, 64 GB memory, 10 Gbps network), generated dummy data with Sqoop and MySQL, and allocated 5 Spark executors with 3 cores and 15 GB memory each to run the job.

We analyzed the runtime logs using CrystalPerf, and it suggests that CPU is the bottleneck resource. We looked into the DAG execution profile and discovered that most CPU entries in the task resource vectors were around 0.785, indicating that CPU had taken the majority of runtime. For further validation, we inspected the profiling traces and the two most frequently called functions were `Text.decode` and `BuiltInZipDecompressor.decompress`, which appeared in 41.18% and 35.29% of sample stacks. Knowing the CPU dominance of the program, the user could simply allocate more CPUs to accelerate it. According to our measurement, doubling the number of CPU cores reduces the job runtime from 25.89s to 14.76s.

Diagnosing Slow Iteration We then turn to an issue [52] from TensorFlow community, where a user trained a neural network over the MNIST [44] dataset. The user already deployed GPUs but still found the iteration time much longer than she had expected. We reproduced the problem on AWS EC2 using a p2.16xlarge instance (16 NVIDIA K80 GPUs, 64 vCPUs, and 732 GB of memory). After taking in the runtime traces, CrystalPerf suggests that I/O sustains for the longest in the application. We verified the conclusion by manually examining the profiling traces and observed that an operator named `QueueDequeueManyV2` (managing queued

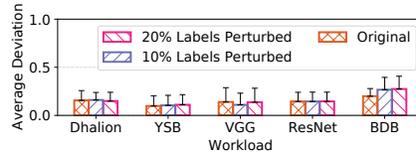


Figure 9: We perturb a portion of the labels used in training two classification models and rerun the prediction experiments on previous workloads. The error bars are *clipped* to show only the upper halves.

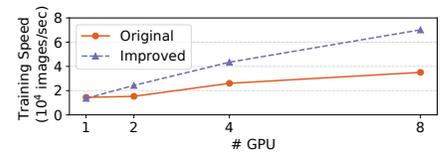


Figure 10: When training models, the user reported the poor multi-GPU scalability in the *original* setting. With CrystalPerf pinpointing the bottleneck as I/O, the user could significantly *improve* it by adopting optimized I/O library.

data in I/O) occupied the majority of iteration time. Such I/O dominance is rare in model training because computation is usually the most time-consuming part, which indicates that the data transfer into and from GPU could be problematic. So we changed GPU to CPU and reran the program. This time, the iteration time dropped from 2 seconds to 80 milliseconds, and the `QueueDequeueManyV2` no longer dominated the traces.

Diagnosing Poor Scalability Next we turn to a question [61] raised in StackOverflow, where a user trained a neural network over the CIFAR-10 [41] dataset. When she increased the number of GPUs, she noticed that the scalability of TensorFlow in the multi-GPU setting was rather poor. We reproduced the user’s problem on AWS EC2, using p3.16xlarge instances (8 Tesla V100 GPUs, 64 vCPUs and 488 GB of memory). After analyzing the traces, CrystalPerf reports that the program is actually I/O bound. Again, we verified this conclusion by checking the traces and noticing that operation `MEMCPYHtD` (copying data from CPU to GPU) took a significant portion of time. As a remedy, we enabled NCCL [53], an optimized library for inter-GPU I/O, and reran the program. Figure 10 shows the increased scalability from the *original* setting to *improved*, demonstrating the effectiveness of CrystalPerf in addressing performance issues.

8 Discussion

Profiling Overhead As runtime logs are enabled by default in many frameworks, the overhead of CrystalPerf mainly comes from the sampling profiler, which we set to sample once per 100ms. We measure Spark applications with and without the profiler and their runtimes differ within 2%, lower than the overhead of instrumentation in SnailTrail [34] (10%). Furthermore, if the framework provides the called functions (e.g., the built-in profiler in TensorFlow), CrystalPerf does not even require profiling.

Extending to New Frameworks In §6 we have introduced the pluggable architecture of our CLI tool, where multiple frameworks can share some common modules such as re-

source inference and basic performance models. When extending it to a new framework, the user has to provide the plugins that parse the generated runtime logs and construct the execution profile. From our experience, it does not involve much labor work. For instance, after implementing the prototype for Spark and TensorFlow, we extend CrystalPerf to Flink with only 172 lines of code.

Advantage over Existing Works As a major improvement, CrystalPerf models dataflow jobs as general DAGs instead of domain-specific structures. For instance, Paleo [62], a performance model for deep learning, is derived from the layer operations of GPUs; Ernest [74] models machine learning pipelines with the common communication patterns specific to machine learning; the stage-by-stage model in Starfish [31] exactly matches the execution of MapReduce jobs. Whereas their model structures limit their applicability scope, CrystalPerf targets a broader array of dataflow computations (e.g., data analytics, stream processing, ML) and exploits commonly available information such as the source code, the execution trace, and the job DAG.

Limitations in Streaming Scenarios Although CrystalPerf is applicable to Flink (§7.1), its ability is restricted by several properties of streaming applications. First, the operators in stream processing can scale up or down as the incoming data fluctuates [40]. Such dynamic parallelism changes the underlying computation graph and thus requires CrystalPerf to update its model accordingly. Second, the processing latency varies with the arrival rate of data [16] as a larger data batch takes a longer time to process. CrystalPerf does not include the rate information for generality. If the users would like to monitor the performance more precisely, we recommend them to record the input rate of data streams and augment the analysis given by CrystalPerf.

9 Conclusion

In this paper, we have presented CrystalPerf, an instrumentation-free, framework-independent approach to performance debugging and reasoning for dataflow computations. For each job, CrystalPerf constructs a DAG execution profile, and infers the resource usage of each operation node from its code and documentation with two machine learning classifiers. We have implemented CrystalPerf as a CLI tool supporting three mainstream frameworks and evaluated it with multiple workloads and use cases. The results show that CrystalPerf can accurately predict the job runtimes under various resource changes and effectively address performance issues.

Acknowledgment

We would like to thank our shepherd, Vasiliki Kalavri, and the anonymous reviewers for their valuable feedback that helps improve the quality of this work. This research was

supported in part by RGC GRF grant 16213120 and ECS grant 26213818. Huangshi Tian and Minchen Yu were supported in part by the Hong Kong PhD Fellowship Scheme and the Huawei PhD Fellowship Scheme, respectively.

References

- [1] Tensorflow benchmarks. <https://github.com/tensorflow/benchmarks>, 2020.
- [2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.
- [3] Alan Akbik, Duncan Blythe, and Roland Vollgraf. Contextual string embeddings for sequence labeling. In *COLING*, 2018.
- [4] akka. Apache Akka. <https://akka.io>, 2019.
- [5] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, et al. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, 2017.
- [6] Gene 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*, 1967.
- [7] UC Berkeley AMPLab. Big data benchmark, 2014. <https://amplab.cs.berkeley.edu/benchmark/>.
- [8] Berne. Spark count() taking long to run. <https://bit.ly/2J7E8ma>, 2017.
- [9] Walter Binder. Portable and accurate sampling profiling for java. *Software: Practice and Experience*, 2006.
- [10] Brad Calder, Ju Wang, Aaron Ogus, Niranjana Nilakanthan, Arild Skjolsvold, et al. Windows azure storage: A highly available cloud storage service with strong consistency. In *SOSP*, 2011.
- [11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, et al. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015.
- [12] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Neural Information Processing Systems, Workshop on Machine Learning Systems*, 2015.
- [13] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, 2016.

- [14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, et al. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *EMNLP*, 2014.
- [15] Google Cloud. Ai platform, 2019. <https://cloud.google.com/ai-platform/>.
- [16] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. Adaptive stream processing using dynamic batch sizing. In *SoCC*, 2014.
- [17] Netty Developers. Netty project. <https://github.com/netty/netty>, 2019.
- [18] Parquet Developers. Parquet mr. <https://github.com/apache/parquet-mr>, 2019.
- [19] Danyel Fisher, Rob DeLine, Mary Czerwinski, and Steven Drucker. Interactions with big data analytics. *INTERACTIONS*, 19(3), 2012.
- [20] Avrielia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: self-regulating stream processing in heron. *VLDB*, 2017.
- [21] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, et al. A configurable cloud-scale dnn processor for real-time ai. In *ISCA*, 2018.
- [22] functional-streams-for scala. Fs2: Functional streams for scala. <https://github.com/functional-streams-for-scala/fs2>, 2019.
- [23] Felix Geisendörfer. fgprof - the full go profiler. <https://github.com/felixge/fgprof>, 2020.
- [24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*, 2003.
- [25] Corvios GmbH. Nprofiler. <https://www.nprofiler.com/>, 2015.
- [26] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, 2016.
- [27] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *OSDI*, 2016.
- [28] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Gollhofer, et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Trans. Storage*, 14(3), 2018.
- [29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.
- [30] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.
- [31] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. In *VLDB*, 2011.
- [32] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. In *VLDB*, 2011.
- [33] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *SoCC*, 2011.
- [34] Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, et al. Snailtrail: Generalizing critical paths for online analysis of distributed dataflows. In *NSDI*, 2018.
- [35] Einar W. Høst and Bjarte M. Østvold. Debugging method names. In *ECOOP*, 2009.
- [36] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, et al. Gray failure: The achilles heel of cloud-scale systems. In *HotOS*, 2017.
- [37] API Specification Java Platform, Standard Edition 7. Class standardsocketoptions. <https://docs.oracle.com/javase/7/docs/api/java/net/StandardSocketOptions.html>, 2020.
- [38] Yurong Jiang, Lenin Ravindranath Sivalingam, Suman Nath, and Ramesh Govindan. Webperf: Evaluating what-if scenarios for cloud-hosted web applications. In *SIGCOMM*, 2016.
- [39] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017.
- [40] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, et al. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *OSDI*, 2018.
- [41] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.
- [42] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, et al. Twitter heron: Stream processing at scale. In *SIGMOD*, 2015.

- [43] Palden Lama and Xiaobo Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *ICAC*, 2012.
- [44] Yann LeCun. The mnist database of handwritten digits, 1998. <http://yann.lecun.com/exdb/mnist/>.
- [45] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, et al. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.
- [46] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in data-parallel systems. In *OSDI*, 2016.
- [47] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, et al. Distributed graphlab: A framework for machine learning and data mining in the cloud. In *VLDB*, 2012.
- [48] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, et al. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.
- [49] Linux Programmer's Manual. Proc filesystem. <http://man7.org/linux/man-pages/man5/proc.5.html>, 2020.
- [50] Linux Programmer's Manual. Tcp protocol. <http://man7.org/linux/man-pages/man7/tcp.7.html>, 2020.
- [51] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *SIGCOMM*. 2019.
- [52] nryant. `reading_data/fully_connected_reader.py` very slow relative to `fully_connected_feed.py`. <https://bit.ly/2kiT2dD>, 2016.
- [53] Nvidia. Nccl library, 2016. <https://github.com/NVIDIA/nccl>.
- [54] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *SOSP*, 2017.
- [55] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *NSDI*, 2015.
- [56] Vivek S Pai, Peter Druschel, and Willy Zwaenepoel. Io-lite: A unified i/o buffering and caching system. In *OSDI*, 1999.
- [57] Adam Paszke, S. Gross, Francisco Massa, A. Lerer, James Bradbury, et al. Pytorch: An imperative style, high-performance deep learning library. 2019.
- [58] Simon Peter, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system as control plane. In *OSDI*, 2013.
- [59] playframework. Play framework. <https://www.playframework.com>, 2019.
- [60] Meikel Poess and Chris Floyd. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record*, 2000.
- [61] A. J. Polk. Scaling performance across multi gpus. <https://bit.ly/2kCMwif>, 2016.
- [62] Hang Qi, Evan R Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. In *ICLR*, 2017.
- [63] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 30(4):65–79, 2010.
- [64] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In *KDD*, 2016.
- [65] scalaz. Scalaz. <https://scalaz.github.io/>, 2019.
- [66] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [67] Richard Socher, Jeffrey Pennington, Eric H Huang, Andrew Y Ng, and Christopher D Manning. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *EMNLP*, 2011.
- [68] V8 Developer Team. Using v8s sample-based profiler. <https://v8.dev/docs/profile>, 2020.
- [69] Huangshi Tian. Crystalperf. <https://github.com/All-less/crystalperf>, 2021.
- [70] Huangshi Tian, Minchen Yu, and Wei Wang. Supplemental materials to crystalperf. <https://www.cse.ust.hk/~weiwa/papers/crystalperf-suppl.pdf>, 2021.
- [71] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, et al. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *EuroSys*, 2016.

- [72] twitter. Twitter util. <https://github.com/twitter/util>, 2019.
- [73] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, et al. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*, 2017.
- [74] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI*, 2016.
- [75] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *ICAC*, 2011.
- [76] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *ASE*, 2016.
- [77] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communication of the ACM*, 2009.
- [78] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. Hierarchical attention networks for document classification. In *NAACL*, 2016.
- [79] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, et al. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.
- [80] Zhibin Yu, Zhendong Bei, and Xuehai Qian. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In *ASPLOS*, 2018.
- [81] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.
- [82] Ce Zhang, Wentao Wu, and Tian Li. An overreaction to the broken machine learning abstraction: The ease. ml vision. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics*, 2017.
- [83] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *NeurIPS*, 2015.
- [84] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on tensor-flow program bugs. In *ISSTA*, 2018.

Controlling Memory Footprint of Stateful Streaming Graph Processing

Pourya Vaziri
School of Computing Science
Simon Fraser University
British Columbia, Canada
pvaziri@cs.sfu.ca

Keval Vora
School of Computing Science
Simon Fraser University
British Columbia, Canada
keval@cs.sfu.ca

Abstract

With growing interest in efficiently analyzing dynamic graphs, streaming graph processing systems rely on stateful iterative models where they track the intermediate state as execution progresses in order to incrementally adjust the results upon graph mutation. We observe that the intermediate state tracked by these stateful iterative models significantly increases the memory footprint of these systems, which limits their scalability on large graphs.

In this paper, we develop memory-efficient stateful iterative models that demand much less memory capacity to efficiently process streaming graphs and deliver the same results as provided by existing stateful iterative models. First, we propose a *Selective Stateful Iterative Model* where the memory footprint is controlled by selecting a small portion of the intermediate state to be maintained throughout execution. Then, we propose a *Minimal Stateful Iterative Model* that further reduces the memory footprint by exploiting key properties of graph algorithms. We develop incremental processing strategies for both of our models in order to correctly compute the effects of graph mutations on the final results even when intermediate states are not available. Evaluation shows our memory-efficient models are effective in limiting the memory footprint while still retaining most of the performance benefits of traditional stateful iterative models, hence being able to scale on larger graphs that could not be handled by the traditional models.

1 Introduction

Streaming graph processing systems [4, 6, 13, 16, 17, 27, 28, 35] aim to deliver real-time results as the graph structure changes via a continuous stream of edge and vertex mutations. These systems are equipped with efficient iterative processing models that are broadly classified into two types: the *stateless iterative model* and the *stateful iterative model*.

Stateless iterative models used in [6, 13, 27, 28] perform regular iterative processing without capturing additional intermediate values that describe the execution history. When the

graph structure mutates (e.g., new edges/vertices get added or old vertices/edges get removed), they either continue the iterative computation without correctly incorporating the impact of mutations on already computed values (i.e., not guaranteeing accuracy of final results) [28]; or, they conservatively deduce the set of values that could be potentially affected due to mutation (using techniques like tag propagation [27]) and recompute those values from scratch.

On the other hand, stateful iterative models used in [16, 17, 35] capture the intermediate state (representing execution history) as computation progresses, often in terms of intermediate values computed for vertices in each iteration. When the graph structure mutates, only the change in values resulting from those mutations are iteratively propagated to correct the intermediate state and compute accurate final results. As expected, stateful iterative models are more efficient in generating correct final results compared to stateless iterative models simply because the stateful models operate on the precise set of intermediate values that get affected by the change. Stateless iterative models, on the other hand, need to be conservative while generating accurate results since they do not capture intermediate values representing the execution history, and hence, they end up demanding much more computation.

While traditional models like [4] maintain the intermediate state in the order of edges for each iteration, recent works [16, 17, 35] capture information in terms of intermediate values computed for vertices, which results in much lesser memory footprint compared to traditional models. In these systems, the amount of intermediate state saved by the stateful iterative models depends on the nature of the graph algorithms they support. For example, models that operate on asynchronous graph analytics algorithms like BFS, shortest paths, connected components, etc. leverage the monotonic relationship of the values in the algorithm to adjust them directly [35], which requires capturing only $O(|V|)$ intermediate state. On the other hand, models that support synchronous graph algorithms like Co-Training Expectation Maximization (CoEM), Collaborative Filtering (CF), etc. capture the

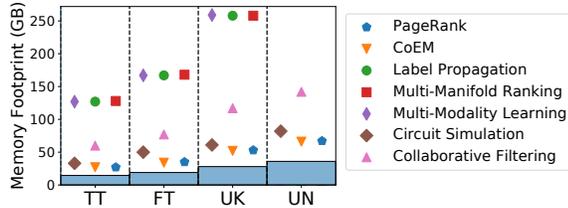


Figure 1: Memory footprint of stateful iterative model across different graph algorithms (shown as different points) and graph datasets (details in Table 2). Solid bars represent the memory consumed by the graph structure.

intermediate state at every iteration in order to incrementally recompute the values iteration-by-iteration and guarantee results equivalent to Bulk Synchronous Parallel (BSP) [32] execution from scratch [16, 17]. Furthermore, the intermediate state in each iteration often contains the aggregation results for vertices along with the vertex values, and their size (in bytes) depends on the graph algorithm being run. For example, CoEM requires 8 bytes per intermediate state of a vertex whereas CF consumes 16 bytes per intermediate state. Other graph algorithms like Multi-Modality Learning (MML) and Label Propagation (LP) have even larger intermediate states depending on the number of labels they operate on.

We profiled GraphBolt [17], a recent state-of-the-art streaming graph processing system, to measure the amount of memory consumed by the intermediate state across different graph algorithms and graph datasets. As shown in Figure 1, the intermediate state consumes at least twice the amount of memory required to hold the input graph itself; for the Collaborative Filtering algorithm this factor increases to over $4\times$ whereas the intermediate state in algorithms like Multi-Manifold Ranking and Label Propagation consumes over an order of magnitude more memory compared to the input graph.

Such high memory footprint significantly limits the scalability of the stateful iterative models on large graphs. For instance we observed that three of the graph algorithms in Figure 1 ran out of memory (with 320GB memory capacity) for the UN graph. With graph datasets growing at a faster rate than memory capacity, it becomes crucial to develop stateful iterative models that do not demand large memory capacities just to hold the intermediate states.

In this paper, we develop novel memory-efficient stateful iterative models that demand much less memory capacity to efficiently process streaming graphs and provide the same BSP guarantees as existing state-of-the-art streaming graph processing systems. First, we propose a *Selective Stateful Iterative Model* where the memory footprint is controlled by selecting a small portion of the intermediate state to be maintained during execution. And then, we propose a *Minimal Stateful Iterative Model* that specializes the incremental processing for certain graph algorithms (depending on their update functions) to drastically reduce the memory footprint.

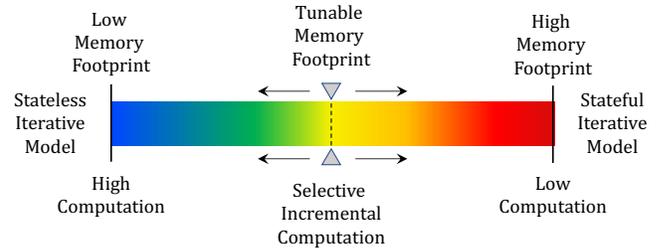


Figure 2: Sliding scale of memory and computation requirements between stateless iterative model and stateful iterative model. By capturing only partial intermediate state (selectively at a fine-grained level), memory consumption can be reduced at the cost of increased computation.

Selective Stateful Iterative Model. The key insight here is that vertex computations within an iteration are independent of each other, and hence, incrementally recomputing the value of a vertex is only dependent on the intermediate state saved for that vertex (i.e., not dependent on the states saved for other vertices). Moreover, the usefulness of different portions of the intermediate state (in terms of the amount of computation pruned out upon graph mutation) is different; this means, intermediate states for certain vertices would end up reducing more computation than those for other vertices.

Based on these insights, our selective stateful iterative model tunes its memory footprint depending on the available main memory by selecting the intermediate vertex states to be captured at a fine-grained level (illustrated in Figure 2). While this enables the model to scale on large streaming graphs, performing incremental computation using the partial intermediate state becomes challenging. This is because the value changes resulting from graph mutations are typically merged with the intermediate vertex states to propagate subsequent (transitive) changes throughout the graph, and how to compute the effects of value changes with missing intermediate states remains unclear. To address this, we develop a *selective incremental processing technique* that correctly computes the effects of changes even when intermediate vertex states are not available. Our strategy captures the nuances of the interaction between vertices with intermediate states and those without intermediate states to ensure that the vertices in the latter set correctly participate in the iteration-by-iteration incremental computation.

Minimal Stateful Iterative Model. In this model, we specialize the incremental processing for certain algorithms. Specifically, algorithms like CoEM and PageRank involve operations that are purely distributive, i.e., outgoing value changes from a given vertex can be directly computed based on the incoming value changes to the vertex. We formalize this characteristic as the novel *distributive update property* to understand the scope of graph computations that are purely distributive. Then, for algorithms that satisfy the distributive

update property, we identify that the effects of graph mutations can be propagated iteration-by-iteration throughout the graph even without using the intermediate states for most of the vertices. Specifically, only the intermediate states for vertices that get directly affected by graph mutation are needed to perform incremental processing over the entire graph.

We use this insight to develop the minimal stateful iterative model, where the amount of intermediate state gets aggressively reduced to a known subset of vertices on which mutations take place, hence consuming a much smaller memory footprint that is dependent on the graph mutations instead of the original graph size. Upon graph mutation, the incremental computation strategy directly operates on changes even for vertices without intermediate states, while also carefully adjusting the available intermediate states for vertices that are directly impacted by mutation.

Results. Our proposed techniques are general, and can be incorporated in any streaming graph processing system to leverage incremental processing without incurring high memory footprint. We implemented both of our proposed memory-efficient models in GraphBolt [17]; since GraphBolt’s existing stateful iterative model maintains intermediate states for all the vertices in the graph, it becomes a natural baseline to demonstrate the effectiveness of our proposed models.

Our evaluation with five real-world graphs and seven synchronous graph algorithms shows that our models demand significantly less memory capacity in comparison to GraphBolt, while still retaining most of its performance benefits. Specifically, the memory footprint of our selective stateful iterative model is dependent on the amount of intermediate state saved; for instance, by selecting only 20% of the intermediate state to be saved, the memory footprint is 35-70% smaller than that for GraphBolt. Furthermore, our minimal stateful iterative model reduces the memory footprint by 28-58% even when it is used for algorithms that have smaller intermediate state to begin with. This allows our memory-efficient stateful iterative models to scale to very large graphs that could not be handled by GraphBolt with the available memory capacity.

2 Background

We briefly review the streaming graph processing model and the stateful iterative processing that performs incremental computation.

Streaming Graph Processing Model. A streaming graph is a graph whose structure keeps on changing via a continuous stream of graph updates (e.g., addition and deletion of vertices and edges). The change in graph structure is also referred to as mutation of graph structure, and each individual update arriving from the stream is also called a mutation. Streaming graph processing systems [4, 16, 17, 27, 28, 35] operate on streaming graphs to continuously produce results consistent with the latest graph structure.

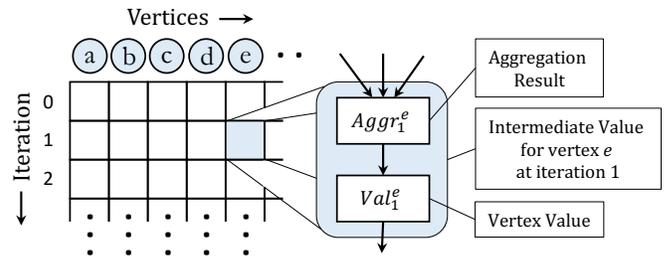


Figure 3: Intermediate state in terms of values relevant for vertices in each iteration. Each intermediate value consists of the aggregation value (to incrementally merge differences) and the vertex value (to compute outgoing differences).

Stateful Iterative Processing Model. *Stateful iterative processing models* used in recent systems like [16, 17, 35] reduce the amount of computation to be performed upon graph mutation using incremental processing. The main idea is to track the intermediate state that captures the necessary details of execution history so that when the graph structure mutates, only the relevant parts of execution history are adjusted or recomputed. To guarantee end results that are same as a Bulk Synchronous Parallel (BSP) execution [32] starting from scratch, GraphBolt employs a *dependency-driven incremental refinement* strategy [17], which incrementally recomputes (or *refines*) intermediate states by propagating changes or *differences* in values resulting due to the graph structure mutation. It propagates the changes in iteration-by-iteration manner, so that correctness guarantees (in form of BSP semantics) are retained for every iteration all the way till the end, hence ensuring that the final result are same as that from a BSP execution from scratch. DZiG [16] (built in GraphBolt) improves the dependency-driven incremental computation by developing a *DelZero-Aware incremental refinement* strategy that retains computation sparsity as iterations progress.

Tracking Intermediate State in Memory. The intermediate state in stateful iterative models is in form of values relevant for vertices at each iteration. As shown in Figure 3, these intermediate values often consist of two components: first, the result of aggregation (also called *aggregation value*) at each vertex that collects values from its incoming edges; and second, the value computed for that vertex. Maintaining both of these values as intermediate state becomes crucial because iterative algorithms often use selective scheduling mechanisms in order to suppress propagation of minor changes (e.g., change less than $1e-2$ threshold). In such cases, the outgoing vertex value for a given iteration may not be based on its aggregation result since the latter can hold multiple minor changes which may not have been propagated to the outgoing neighbors. By explicitly tracking the two values, differences can be correctly computed and propagated based on values visible to the neighbors.

As expected, tracking this intermediate state increases the

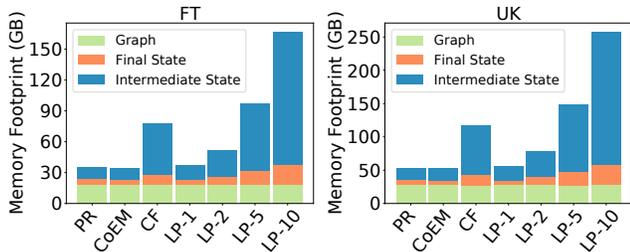


Figure 4: Memory consumption of different components in stateful iterative model: graph structure, final vertex results, and intermediate state. The intermediate state requires different amount of memory across different graph algorithms. On Label Propagation, the memory consumed by intermediate state increases as number of labels increase (indicated by LP- k where k is the number of labels).

memory footprint of such incremental processing techniques. While Figure 1 shows the high memory footprint for different graph algorithms on graph datasets, Figure 4 compares the size of intermediate state relative to the remaining memory consumption of the process (majority of which is taken by the input graph structure and then vertex frontiers). Even though PageRank and CoEM operate on scalar values, their intermediate state and final state consumes nearly as much memory as the remainder of the process. Collaborative Filtering operates on two factors per vertex (i.e., its vertex state is a size-2 vector), and hence, its intermediate state ends up consuming up to 80% additional memory compared to the stateless execution. Finally, Label Propagation operates on feature vectors; as shown in Figure 4, increasing the number of features directly increases the amount of memory consumed by the intermediate state. In fact, maintaining intermediate states with 10 features requires 129GB additional memory for FT and 198GB additional memory for UK, which increases the memory footprint by 3.44 \times and 3.3 \times respectively compared to their stateless executions. Such high amount of memory consumption significantly limits the applicability of the stateful iterative processing model on large graphs.

3 Selective Stateful Iterative Model

3.1 Intuition & Main Idea

Maintaining intermediate state essentially allows incremental processing where the effects of graph mutation are propagated in form of value changes throughout the graph. On the other extreme when intermediate state is not maintained, vertex values that are recomputed in a given iteration have to be pushed out since there is no way to determine whether the new values are different from ones computed prior to graph mutation. We observe that every single vertex computation, either in incremental manner with intermediate state or from scratch without intermediate state, is a local computation. This

means the value for a given vertex can be computed as long as the right values arrive from its in-neighbors (either in form of value differences or actual values). Hence, we selectively trade off the benefits of incremental computation with reduced memory footprint at a fine-grained level.

Our selective stateful iterative model tracks the intermediate states of only a subset of vertices instead of all the vertices in the graph. For vertices whose intermediate states are not tracked, the model reconstructs their states on-the-fly so that changes resulting from graph mutation can be directly propagated. As illustrated in Figure 2, this allows us to limit the memory footprint by directly controlling the subset of vertices whose intermediate values are tracked, at the cost of performing more computation for vertices whose intermediate states are not tracked.

For simplicity, the vertices whose intermediate states are tracked are called *tracked vertices* whereas the remaining vertices are called *untracked vertices*.

3.2 Tracking Useful Vertex States

In order to selectively track intermediate states, we need to answer two main questions: first, how many vertices should be tracked, and second, which specific vertices should be tracked.

A) How many vertices should be tracked? To maximize the benefits of stateful incremental processing, tracking as many vertices as possible becomes an ideal choice. Hence, we can compute the number of tracked vertices based on the memory capacity (or budget) assigned for the process. Specifically, the size of the intermediate vertex states (which is algorithm dependent) can be determined during initialization, which can be used to bound the number of vertices to be tracked using the available memory budget:

$$mem_budget \geq k \times state_size \times t + base_size$$

where k is the number of vertices whose states are tracked, mem_budget is the available memory capacity, $state_size$ is the size of intermediate vertex state, t is the number of iterations for which intermediate state should be captured, and $base_size$ is the memory consumed by other data structures in the system (e.g., input graph structure, vertex frontiers, stream buffers, etc.) along with additional capacity for the graph structure to grow over time. Hence, the above equation can be rewritten to maximize the number of tracked vertices k as:

$$\operatorname{argmin}_k |mem_budget - (k \times state_size \times t + base_size)|$$

Note that the majority of $base_size$ is consumed by the graph data structure, whereas the remaining structures like vertex frontiers (which are simply boolean arrays) often consume less than 10% memory compared to the graph data structure.

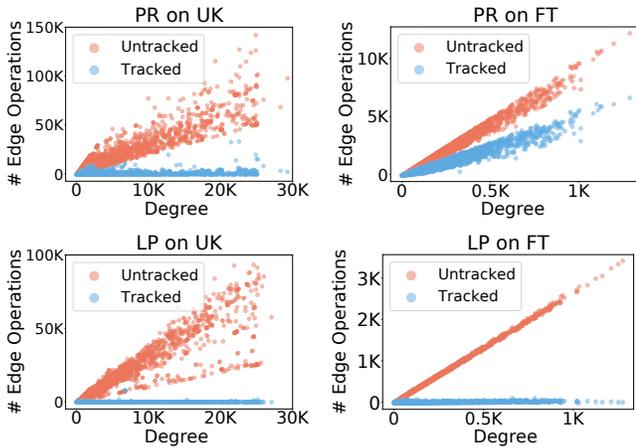


Figure 5: Number of edge operations performed for tracked and untracked vertices based on their in-degrees. Tracking high in-degree vertices reduces more edge operations compared to tracking low in-degree vertices.

B) Which vertices should be tracked? A naive way to select vertices to be tracked can be using random sampling, where tracking of intermediate states can be enabled for a random subset of vertices. While such a strategy easily allows selecting vertices, it remains oblivious of how incremental processing gets performed, and hence it fails to maximize the benefits of incremental processing. Since different vertices require different amount of computation depending on how values get propagated throughout the graph, we must ideally select those vertices that demand high computation so that most of their computation can be effectively eliminated by incremental processing. To do so, we consider the ‘usefulness’ of the intermediate state for each vertex, where the usefulness of an intermediate state is informally defined as the amount of computation it ends up reducing for that vertex. The usefulness of an intermediate state depends on several dynamic factors including the distance (in terms of number of hops) from the vertices where mutations got applied, and the sensitivity of the graph algorithm to changes in graph structure. Since accurately computing such a metric is infeasible for the general case, we approximate the usefulness of an intermediate state using a vertex-local heuristic.

To develop our heuristic, we profiled the amount of computation performed on each vertex when processing a given graph snapshot. The computation for each vertex is measured in terms of the number of edge operations performed for that vertex; this is because edge operations are expensive (often involve atomic writes and random accesses) and they are the primary candidates that incremental processing attempts to reduce [17]. Figure 5 correlates the number of edge operations for different vertices with their in-degrees for two executions: first, the execution where all vertices’ intermediate states are tracked (i.e., GraphBolt’s dependency-driven incremental processing); and second, the execution where computation is started from scratch (i.e., no intermediate state

is tracked). As we can see vertices with higher in-degree demand more computation when their intermediate values are not tracked, and tracking their values reduces their computation requirements. For instance, the top 20% of the high in-degree vertices contribute to up to 34.1-94.9% of the total number of edge operations in Figure 5. Hence, tracking the intermediate states for high in-degree vertices is most useful. On the other hand, the savings for low in-degree vertices are small (visible from the gap between orange points and blue points for low in-degree vertices) since those vertices demand fewer computation even when their intermediate states are not tracked. Therefore, we track the intermediate states for top- k vertices ranked with highest in-degree.

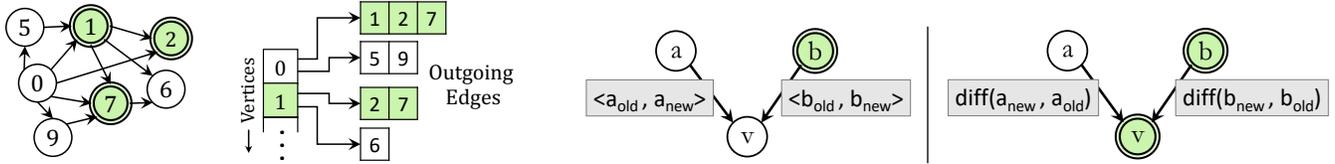
Efficiently Maintaining Tracked Vertex Set. When the graph snapshot gets initialized, a linear pass is performed over the vertices, and the vertex ids are maintained in a degree-ordered max-heap. The vertex ids whose intermediate states must be tracked are then selected in linear time from the heap.

As execution progresses and graph structure mutates, the heap is incrementally adjusted to reflect changes in vertex orderings based on their degree changes. Whenever graph mutations significantly impacts the top- k vertex ranking, the subset of tracked vertices can be refreshed to eliminate certain vertices and add new vertices. As the set of tracked vertices gets updated, the intermediate state for vertices that get newly added to the tracked set needs to become available for subsequent incremental processing. This is achieved during incremental refinement by keeping those newly added vertices active during the iterations so that their intermediate states get incrementally computed, which are then tracked. Vertices that get removed from the tracked set are handled by simply turning off their tracking during the next incremental refinement, and releasing the memory allocated to their tracked state.

3.3 Incremental Processing upon Mutation

With intermediate states available for only a subset of vertices, propagating changes resulting from graph mutations becomes challenging. This is because values during the incremental refinement stage can flow across different vertices regardless of whether their intermediate states are tracked or not. Since we aim to guarantee BSP semantics (similar to systems like [17]), computation of vertex values cannot be deferred as they need to happen in an iteration-by-iteration manner.

We develop a *selective incremental processing* technique that operates on selective intermediate states, i.e., where a selected subset of vertices are tracked. Our technique effectively separates out the interactions between tracked and untracked vertices so that right values get propagated across the edges depending on whether their source and target vertices are tracked or untracked. We first summarize how the graph layout can be optimized when selective intermediate states are maintained, and then discuss the details of our selective incremental processing.



(a) Optimized graph layout. Vertices 1, 2 and 7 are tracked (highlighted), and remaining vertices are not tracked. Each vertex maintains two adjacency lists for outgoing edges: one for tracked neighbors and other for untracked neighbors.

(b) Values propagated based on whether the target vertex is tracked (highlighted) or untracked. Untracked vertices receive the old and new values, whereas tracked vertices directly receive difference in values.

Figure 6

Optimizing Graph Layout. Streaming graph processing systems use efficient dynamic graph representations to handle rapid graph mutation and enable efficient parallel operations on active edges and vertices [6, 7, 15, 17]. Since our selective incremental processing handles the interactions for tracked vertices in a different manner compared to the interactions for untracked vertices, the graph layouts can be improved to avoid expensive checks while propagating values during the refinement process. Specifically, the edges between tracked vertices and untracked vertices can be separated out in the graph layout itself; by doing so, all computations on edges whose target vertices are either tracked or untracked can be performed directly in form of parallel operations without verifying whether every individual target vertex is tracked or untracked.

Since we incorporate our technique in GraphBolt, which uses adjacency lists to hold graph snapshots, such a separation results in the graph layout shown in Figure 6a. Here, each vertex now holds two vectors for its outgoing neighbors: the first vector contains edges whose targets are tracked vertices, and the second vector contains edges whose targets are untracked vertices.

Propagating Differences upon Graph Mutation. When the graph structure mutates, the incremental computation must correctly propagate changes throughout the available intermediate states to compute final results. To retain BSP guarantees at the end of each iteration, our selective incremental processing propagates values iteration-by-iteration.

With only a subset of intermediate states available, processing for different vertices is handled differently. For tracked vertices, the intermediate states are incrementally refined in-place as processing progresses through iterations. Whereas for untracked vertices, a single vector is maintained to hold their latest values as processing progresses through iterations (similar to computing from scratch without intermediate states). In each iteration, the values propagated across edges are based on whether the target vertices are tracked or untracked, as shown in Figure 6b. If the target vertex is tracked, then the difference in value is propagated along its incoming edge similar to the traditional dependency-driven incremental processing [17]. On the other hand, if the target vertex is untracked, then both the old value (from before graph mutation) and the new value (resulting from graph mutation) are propagated along the edge.

This allows the target vertex to compute the necessary differences for its outgoing neighbors in the subsequent iteration.

Algorithm 1 shows how our selective incremental processing propagates values upon graph mutation. In each iteration, the differences directly resulting from graph mutation are propagated (lines 6-13), and then the resulting differences are propagated for the tracked and untracked vertices (line 15-22). The tracked vertices acquire the difference between the previous value change and the new value change. The untracked vertices, on the other hand, receive two values: the previous change and the new change. This allows the untracked vertices to recompute the old aggregation along with the updated aggregation. Once the values arrive at the required active vertices, their vertex values are computed to identify the differences to be propagated in the next iteration (lines 24-33).

As we can see on lines 15 and 19, operations on edges based on their target being tracked or untracked are directly invoked in parallel without any checks per edge, mainly because of the optimized graph layout described above that separates the edges. Moreover, since the selective incremental processing recomputes the vertex values to identify differences, our model tracks only the aggregation values as intermediate states (i.e., it does not track the intermediate vertex values, which is also maintained as part of intermediate state in the traditional dependency-driven incremental refinement [17]).

4 Minimal Stateful Iterative Model

In this model, we aggressively eliminate the tracking of intermediate state by specializing the incremental processing for certain graph algorithms. Specifically, our model will directly operate on value differences without reconstructing the intermediate states so that effects of mutations get propagated only as value differences throughout the iterations.

We first formalize the *distributive update property* that enables this specialization, and then discuss the details of the minimal stateful iterative model.

4.1 Distributive Update Property

Computations in graph algorithms can be modelled as:

$$val(v) = A \left(\bigoplus_{(u,v) \in E} (S(val(u))) \right)$$

Algorithm 1 Selective Incremental Processing

```

1: par for  $e \in E$  s.t.  $\text{target}(e)$  is untracked do
2:   Activate  $\text{source}(e)$  for propagation to untracked targets
3: end par for

4: for  $i \in [1..]$  do
5:   /* Propagate changes directly resulting from mutations */
6:   par for  $e \in$  mutated edges s.t.  $\text{target}(e)$  is tracked do
7:     Propagate old change if  $e$  is added; otherwise retract old change
8:     Activate  $\text{target}(e)$  for vertex computation
9:   end par for
10:  par for  $e \in E$  mutated edges s.t.  $\text{target}(e)$  is untracked do
11:    Propagate old change if  $e$  is removed; otherwise retract old change
12:    Activate  $\text{target}(e)$  for vertex computation
13:  end par for

14:  /* Propagate transitive changes from active vertices */
15:  par for  $e \in E$  s.t.  $\text{target}(e)$  is tracked and ( $\text{source}(e)$  is active or  $e$  is
16:  mutated edge) do
17:    Propagate difference between old change and new change
18:    Activate  $\text{target}(e)$  for vertex computation
19:  end par for
20:  par for  $e \in E$  s.t.  $\text{target}(e)$  is untracked and  $\text{source}(e)$  is active do
21:    Propagate old change and new change
22:    Activate  $\text{target}(e)$  for vertex computation
23:  end par for

23:  /* Compute vertex values and differences to push in next iter */
24:  par for  $v \in$  active tracked vertices do
25:    Merge difference in  $v$ 's intermediate state
26:    Compute  $v$ 's old value and new value
27:    Activate  $v$  for propagation if difference in value changes is not  $\emptyset$ 
28:  end par for
29:  par for  $v \in$  active untracked vertices do
30:    Merge old change in  $v$ 's old value and new change in  $v$ 's new value
31:    Compute  $v$ 's old value and new value
32:    Activate  $v$  for propagation if difference in value changes is not  $\emptyset$ 
33:  end par for
34: end for

```

where \oplus is the aggregation function that combines incoming values to a vertex, S is the function that transforms the source's value to be aggregated (analogous to scatter operation in [8]), and A is the vertex function that computes the vertex value using the aggregation result. For instance, in PageRank \oplus is the *sum* operation, S is the function that divides the rank value with outdegree (i.e., $\text{pr}(u) / \text{out_degree}(u)$), and A is the linear equation that computes rank value using the result of \oplus and damping factor (i.e., $(1-d) + d * \text{sum}$).

The *distributive update property* states that the computation of vertex value can be distributed as sub-computations over its incoming neighbors' values, i.e.,

$$A\left(\bigoplus_{k \in \{w,x,y,z\}} (S(k))\right) = \gamma_{k \in \{w,x\}, \{y,z\}} \left(\alpha\left(\bigoplus_{k' \in k} (S(k'))\right)\right)$$

where γ and α are functions derived from A . This property is important because it allows directly computing the difference in the target value from the difference in the source value without reconstructing $\bigoplus(S(*))$. This allows our minimal stateful iterative model to aggressively reduce the intermediate state by simply not tracking the results from \bigoplus . For instance, in our PageRank example if a source's rank value

Graph Algorithm	Distributive Update Property	Reason for Violation
PageRank	✓	-
Co-Training Expectation Maximization	✓	-
Katz Centrality	✓	-
Path-based / Monotonic Algorithms like: Breath First Search, Shortest Paths, Connected Components, Widest Paths, Minimal Spanning Tree	✓	-
Collaborative Filtering	✗	Matrix Inverse & Multiplication
Circuit Simulation	✗	Division
Label Propagation	✗	Value Normalization
Multi-Manifold Ranking	✗	
Multi-Modality Learning	✗	

Table 1: Algorithms whose computations satisfy (✓) or violate (✗) the distributive update property.

changes from u_1 to u_2 , then the change in value of the destination vertex v gets directly computed as $d * (u_2 - u_1) / \text{out_degree}(u)$. Note this does not require explicitly reconstructing the sum variable for v .

Applicability. The distributive update property is different from just the aggregation operation being distributive that is studied in static graph processing [39]. While most of the aggregation operations are distributive (which enables edge parallel incremental operations, as well-known in prior research), the distributive update property also requires the vertex functions to be distributive. For instance, the PageRank computation satisfies this property since its linear equation only operates on the aggregation value and constants, and hence, any change in rank value of source vertex can be directly incorporated in the destination value. On the other hand, even though algorithms like Collaborative Filtering [40] and Multi-Modality Learning [31] have *sum* aggregation (which is distributive), their vertex functions are not distributive since they involve operations like *normalization* and *matrix inverse*. Table 1 classifies various graph algorithms based on whether they satisfy or violate the distributive update property.

4.2 Tracking Minimal Vertex State

While the differences can be propagated without computing the intermediate states at each iteration, these differences need to be grounded w.r.t. some basis so that they are meaningful. Hence, we track the earliest intermediate state that initiates the incremental computation when graph structure mutates. These earliest intermediate states correspond to the states of the mutation points (e.g., vertices whose edges got mutated)

Algorithm 2 Incremental Processing with Minimal State

```
1: for  $i \in [1..n]$  do
2:   /* Propagate old values */
3:   par for  $e \in$  mutated edges do
4:     Propagate old value if  $e$  is added; otherwise retract old value
5:     Activate target( $e$ ) for vertex computation
6:   end par for

7:   /* Propagate transitive changes from active vertices */
8:   par for  $e \in E$  s.t. source( $e$ ) is active or  $e$  is mutated edge do
9:     Propagate change
10:  end par for
11:  par for  $v \in$  active vertices do
12:    if  $v$  is tracked then
13:      Merge change in  $v$ 's intermediate state
14:    end if
15:    Activate  $v$  for propagation if change is not  $\emptyset$ 
16:  end par for
17: end for

18: /* Compute final vertex values */
19: par for  $v \in V$  do
20:   Merge change in vertex value with old vertex value
21: end par for
```

since those points start propagating the changes directly based on the specific edge/vertex that gets added/deleted. Apart from the earliest states, no other intermediate state is captured because the computations purely operate on differences to propagate through the rest of the iterations.

Hence, the potential mutation points in the graph are the subset of vertices whose intermediate states are tracked. These vertices are identified based on application-specific insights like mutations occurring at certain important vertices, or what-if queries based on certain regions of the graph. When potential mutation points cannot be identified a priori, high in-degree vertices can be tracked (similar to Section 3.2) in order to increase the chances of mutations getting applied to tracked vertices. As we see next, the incremental processing can automatically handle the case when mutations occur on vertices that are not tracked as well.

4.3 Incremental Processing

With the distributive update property, the impact of mutations on tracked vertices can be computed directly using difference in values. When graph structure mutates, incremental computation is performed in iteration-by-iteration manner by purely operating on differences. Algorithm 2 shows how the differences are identified and propagated. Unlike the selective incremental processing technique, the distributive update property enables straightforward propagation of differences. The mutated edges propagate and retract the old values (lines 3-6), and their target vertices compute the differences. These differences are further propagated in subsequent iterations (line 9). If the target vertex is tracked, the differences are merged in the intermediate state as computation progresses. In the end, the cumulative differences are incorporated with the vertex values to generate the final result (line 20).

When mutation occurs on vertices that are not tracked, the processing can dynamically switch to selective incremental processing (Section 3.3) that correctly propagates value differences based on whether the target vertices are tracked or untracked. This switch happens seamlessly without performing additional computation since the selective incremental processing reconstructs the missing intermediate states on-the-fly as it progresses iteration-by-iteration.

5 Evaluation

We thoroughly evaluate our memory-efficient stateful iterative models and compare their performance with the dependency-driven incremental processing model from GraphBolt [17] (which delivers high performance at the cost of high memory consumption). Specifically, we answer the following questions:

1. How effective is our selective stateful iterative model in controlling the memory footprint?
2. How does the performance of our selective stateful iterative model vary as the number of vertices being tracked changes?
3. How effective is our minimal stateful iterative model in maintaining a small memory footprint while still delivering high performance?
4. How do our memory-efficient stateful iterative models perform when processing a large number of simultaneous graph mutations?

5.1 Implementation Details

We implemented our memory-efficient stateful iterative models in the GraphBolt system for two main reasons: first, it allows our models to utilize the efficient implementation of the underlying framework (e.g., parallelization strategy, atomics, frontiers, etc.); and second, it enables direct performance comparison of our models with GraphBolt's existing execution model.

We implemented the optimized graph layout for the adjacency list representation (discussed in Section 3.3) to replace the existing adjacency list data structure. The intermediate states for selected subset of vertices are tracked in their respective arrays. The state arrays get allocated vertically (per vertex) instead of horizontally (per iteration) so that arrays for untracked vertices are not allocated (hence reducing memory footprint), while at the same time the intermediate states for tracked vertices get addressed without using any hashmap.

5.2 Experimental Setup

We use seven synchronous graph algorithms. PageRank (PR) [23] computes the importance of web-pages based on incoming links to those pages. Collaborative Filtering (CF) [40] is a context-based technique used in recommender systems

Graph	Vertices	Edges	Graph Size	
			Without Final State	With Final State
TwitterMPI (TT)	52.6M	2B	14.2GB	19-30GB
Friendster (FT)	68.3M	2.5B	18.74GB	24-38GB
UK-2007-05 (UK)	105M	3.7B	27.75GB	36-59GB
UK-union (UN)	133M	5.5B	36.2GB	48-80GB
Clueweb (CWB)	978.4M	42.5B	130GB	197-240GB

Table 2: Real-world graphs used in experiments [1, 2, 3]

to classify associated items while Co-Training Expectation Maximization (CoEM) [22] is a semi-supervised learning algorithm for named object identification. Multi-Manifold Ranking (MMR) [37] is a ranking method that uses multiple image manifolds each constructed using a different image features. Multi-Modality Learning (MML) [31] and Label Propagation (LP) [41] are learning algorithms that disperse labels from a subset of vertices to assign label to the rest of the graph. Circuit Simulation (CS) [12] simulates flow in a circuit by solving partial differential equation.

The LP, MMR and MML algorithms compute vector of features for each vertex, whereas the remaining algorithms operate on scalar vertex values except for CF which operates on two factors per vertex. Computations in PageRank and CoEM follow the distributive update property (described in Section 4.1), and hence we evaluate our minimal stateful iterative model with these two benchmarks. Computations in the remaining benchmark do not follow the distributive update due to the complex sub-operations they involve: specifically, LP, MMR and MML normalize the feature vectors in every iteration, CF computes matrix inverse, and CS uses division on its aggregation values.

Table 2 lists the six real-world input graphs used for evaluation. Similar to [17, 28], we obtained an initial fixed point when 50% of edges were loaded, and streamed in the remaining edges to model edge insertions, while randomly sampled edges from the loaded graph were used for edge deletions. To eliminate the effects of locality, we shuffled the edges while forming our edge streams. Starting with 50% of edges is not a requirement for GraphBolt and other systems (i.e., one can start with an empty graph as well), but doing so allows us to evaluate the common scenario where a graph snapshot is already present and mutations are streamed in. For CWB graph which is significantly large, initial fixed point was obtained with 7% of edges so that at least stateless execution could successfully execute. The algorithms that operate on vectors consume high memory, and as expected, the memory footprint increases as the vector size increases. Unless otherwise stated, we use 10 features for LP, MMR and MML so that the GraphBolt baseline could hold the intermediate state without running out of memory, and 2 features are used for CWB graph to ensure that stateless execution could successfully execute. Similar to [17], we run all algorithms for 10 iterations. Unless otherwise stated, we apply 10K edge mutations to evaluate how quickly our models compute the final result; we also vary the mutation batch size from a single mutation

to up to 10 million edge mutations.

All experiments were performed on Oracle Cloud VM.Standard2.24 shape containing Intel(R) Xeon(R) 8167M processor with 24 physical cores (48 threads) and 320GB main memory running 64-bit Ubuntu 18.04.

Throughout the evaluation, we use the following notations for different executions:

- **Selective- $k\%$** : this is our selective stateful iterative model that tracks $k\%$ of total vertices.
- **Minimal**: this is our minimal stateful iterative model.
- **GraphBolt**: this is GraphBolt’s execution [16, 17], which tracks the intermediate state for all vertices.
- **Stateless**: this baseline does not track any intermediate state, and recomputes values from scratch upon graph mutation (same as GB-Reset in [17]).

5.3 Selective Stateful Model Performance

Figure 7 shows the memory footprint and execution time for our selective stateful iterative model when tracking the intermediate state for 20%, 40%, 60% and 80% of the vertices. The figure also compares the performance with GraphBolt and stateless executions. Figure 8 summarizes similar results for CWB graph. As we can see, our selective stateful iterative model is effective in controlling the memory footprint by tracking only the selected subset of vertices. For instance, it consumes 35-70% less memory than GraphBolt when tracking only 20% of vertices, while at the same time delivering 15-83% of the performance gains provided by GraphBolt over the stateless execution. In fact, GraphBolt runs out of memory for certain cases while the selective executions end up successfully executing and delivering high performance.

By tracking more intermediate states the memory footprint increases and the execution time decreases mainly because the intermediate state helps in incremental refinement; this is visible as decreasing number of edge operations in Figure 9 for FT graph as the number of tracked vertices increases from 20% to 80%. Since stateless execution does not track any intermediate state, its memory footprint is the lowest while the execution time is highest; on the other extreme, since GraphBolt tracks intermediate state for all vertices, its memory footprint is the highest while its execution time is the lowest. Our selective stateful iterative model trades off memory footprint for more computation, and hence delivers performance between the two extremes.

We observe that the increase in memory footprint from stateless to selective-20% is higher compared to the increase between consecutive selective variants (e.g., between selective-20% and selective-40%). This is because the selective incremental processing computes both the new value (after mutation) and the old value (prior to mutation) so that differences can be propagated from untracked values; holding

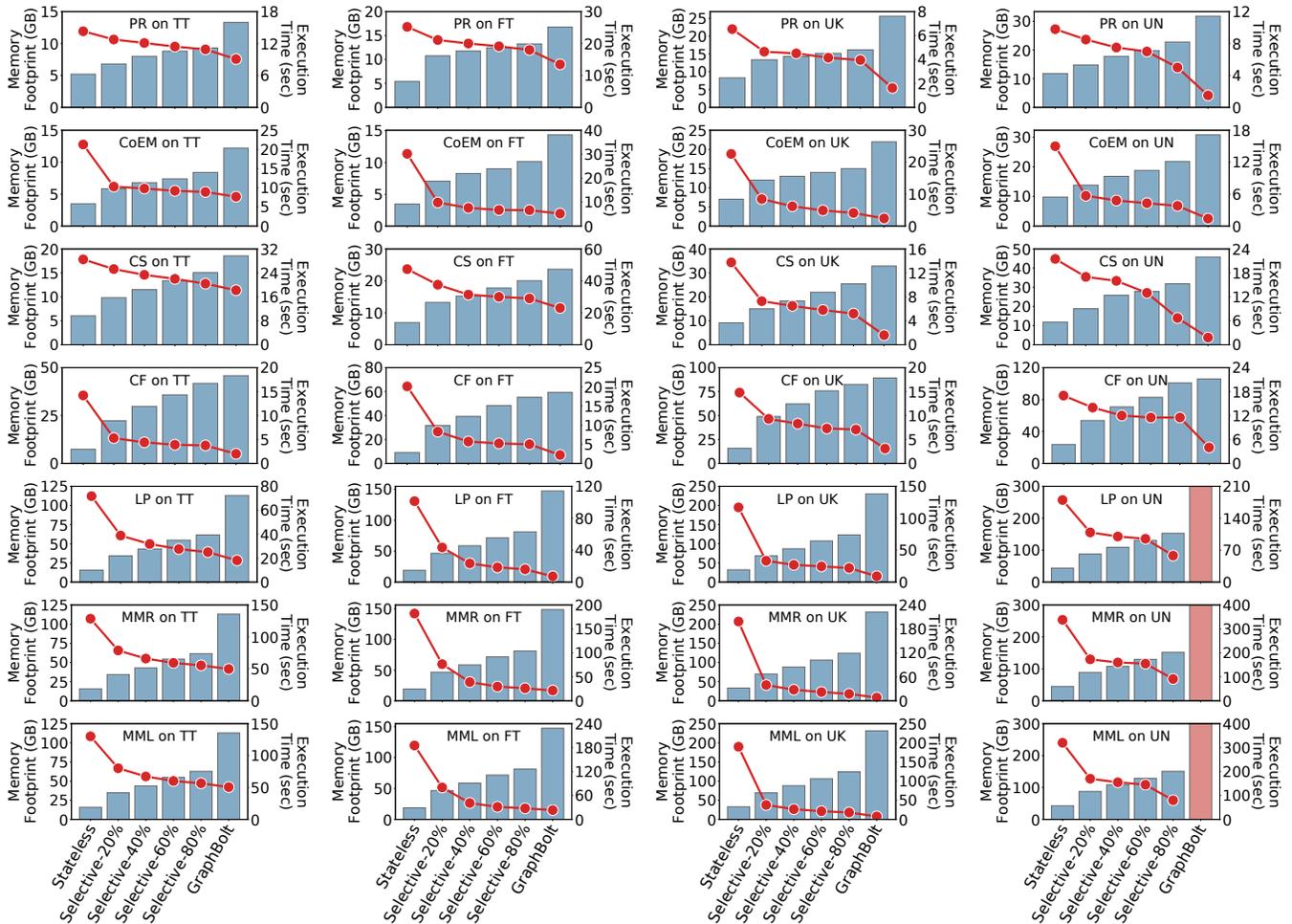


Figure 7: Performance of our selective stateful iterative model compared to the stateless iterative model and the stateful iterative model from GraphBolt. The memory footprints (in GB) are shown as bars (left y-axis) and the execution times (in seconds) are shown as points (right y-axis). Red bar indicates the execution ran out of memory.

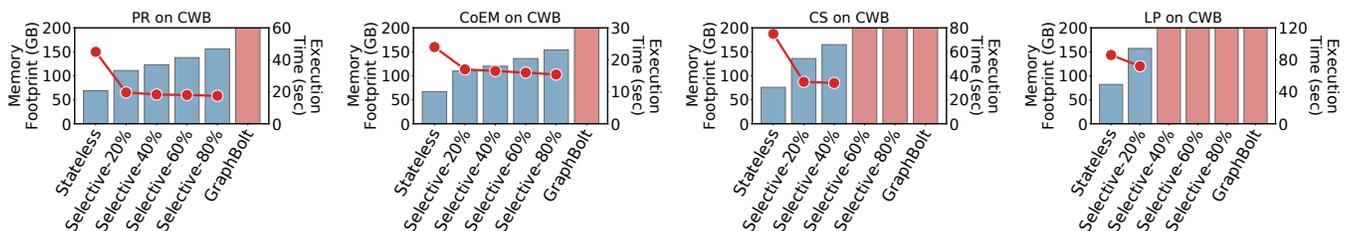


Figure 8: Performance of our selective stateful iterative model on CWB graph compared to the stateless iterative model and the stateful iterative model from GraphBolt. The memory footprints (in GB) are shown as bars (left y-axis) and the execution times (in seconds) are shown as points (right y-axis). Red bar indicates the execution ran out of memory.

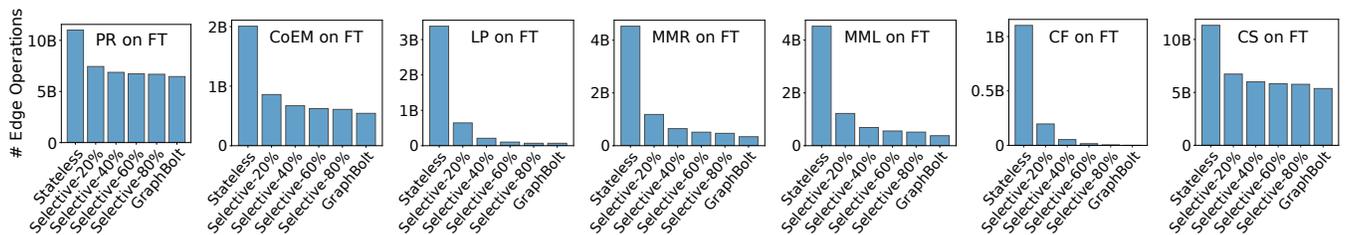


Figure 9: Number of edges operations executed by our selective stateless iterative model, compared to the stateless iterative model and the stateful iterative model from GraphBolt.

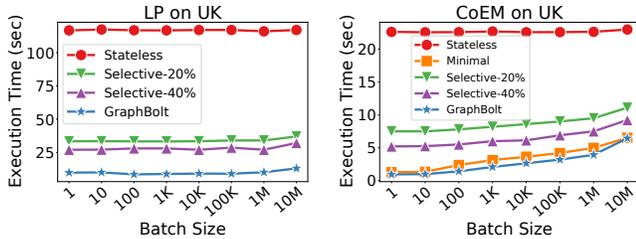


Figure 10: Execution times (in seconds) for different iterative models across varying number of mutations per batch.

these values in memory adds to the memory footprint, which is an overhead that stateless executions do not incur.

For a given graph, the memory footprint depends on the size of intermediate state in the benchmark. Hence, for each graph the memory footprints are lower for PageRank and CoEM since they operate on scalar values, while the footprints are higher for LP, MMR and MML due to their use of feature vectors. This is also the reason why GraphBolt runs out of memory for only LP, MMR and MML on UN graph; whereas on CWB graph only certain executions of selective variants run successfully.

The execution time, on the other hand, is dependent on how the values propagate across iterations which is dependent on the graph algorithm and the structure of input graph. Therefore, the performance benefit with saving selective intermediate state is different for different cases. For instance, selective-20% is $5\times$ faster compared to stateless for MMR on UK, whereas it is only $1.63\times$ faster for MMR on TT. On the other hand, selective-20% is $2.4\times$ faster compared to stateless for CF on FT, but only $1.2\times$ faster for CF on UN.

For most of the cases, we observe that tracking intermediate state for only 20% of vertices achieves 15-83% of performance gains that are achieved by GraphBolt. For instance selective-20% on MMR achieves $1.6-5\times$ compared to stateless executions across different graphs, whereas GraphBolt achieves $2.6-24.4\times$ compared to stateless execution. This is mainly because our model tracks the high-degree vertices that demand more computation if their states are not available, and hence incremental processing for those high-degree vertices ends up achieving high performance benefits. This is a benefit especially for skewed graphs since the memory consumption for selective-20% is much less compared to GraphBolt, while at the same time the performance gains are high. As expected, the performance gains from incremental processing reduce as the number of tracked vertices increases.

Finally, for cases like CF we observe that the difference in execution times between selective-80% and GraphBolt is higher compared to the difference between consecutive selective variants (e.g., between selective-60% and selective-80%). This is again because the selective incremental processing computes both the old values and the new values whereas GraphBolt directly computes the value changes. The effects of these additional computations become more visible for CF

since it has relatively expensive vertex computations (involving a matrix inverse operation).

Scaling with Mutation Batch Sizes. Figure 10 shows the performance of our selective stateful iterative model with 20% and 40% tracked vertices as the mutation batch size increases from a single mutation to up 10 million edge mutations. The memory footprints of the stateful iterative models remain same as in Figure 7 since they are mainly dependent on the number of tracked vertices. However, we observe that the amount of computation performed increases as more mutations get simultaneously applied. Hence, the execution time for both, selective stateful model as well as GraphBolt increases as mutation batch size increases. Since the stateless execution simply recomputes from scratch, its execution time increases very slowly across different mutation batch sizes.

5.4 Minimal Stateful Model Performance

Figure 11 shows the memory footprint and execution times for our minimal stateful iterative model along with GraphBolt and stateless executions on PageRank and CoEM. Since the minimal stateful iterative model only tracks the earliest intermediate states that form the basis for differences, its memory footprint is $1.4-2.4\times$ smaller compared to GraphBolt and only $1.1-1.3\times$ higher than stateless execution. Moreover, the incremental computation in the minimal stateful iterative model purely operates on value differences, and hence, delivers high performance. Our minimal stateful iterative model is $1.1-8.2\times$ faster than stateless executions, which results in 65-90% of the benefits delivered by GraphBolt.

Scaling with Mutation Batch Sizes. Figure 10 shows the performance of the minimal stateful iterative model for CoEM as the mutation batch size increases from a single mutation to up 10 million edge mutations. Similar to selective model, the execution time for the minimal stateful iterative model increases as the number of simultaneous mutations increases; nevertheless, it is $3.5-17.4\times$ faster than the stateless execution.

Unlike the selective model, the memory footprint of our minimal stateful iterative model is sensitive to the number of updates. As the mutation batch size increased from 1 to 10M, the memory footprint of our minimal stateful iterative model increased from 0.5GB ($1.06\times$ higher than stateless, and $2.9\times$ lower than GraphBolt) to 2GB ($1.3\times$ higher than stateless, and $2.4\times$ lower than GraphBolt).

6 Related Work

Several dynamic graph processing systems have been developed in literature. We first discuss the systems with stateful iterative models, and then summarize the works that use stateless models.

Kickstarter [35], GraphBolt [17] and DZiG [16] develop efficient streaming graph processing solutions using stateful iterative models for incremental computation. Upon graph

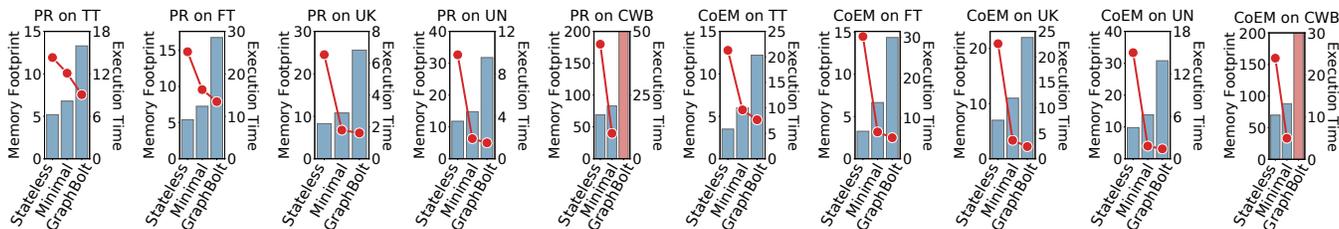


Figure 11: Performance of our minimal stateful iterative model compared to the stateless iterative model and the stateful iterative model form GraphBolt. The memory footprints (in GB) are shown as bars (left y-axis) and the execution times (in seconds) are shown as points (right y-axis). Red bar indicates the execution ran out of memory.

mutation, these systems use the intermediate state to quickly adjust the computed values and deliver final results corresponding to the latest graph version. KickStarter [35] focuses on graph algorithms like BFS and SSSP that use monotonic functions. Its runtime exploits the monotonic relationship between vertex values to capture dependencies between the latest computed values, resulting in only one intermediate state per vertex. Hence, its memory footprint does not drastically increase compared to stateless execution of those algorithms as the input graph consumes the most amount of memory. GraphBolt [17] and DZiG [16] focus on the broader class of graph algorithms that run in BSP manner. They capture the dependency information across intermediate vertex values as computation progresses, and not just across the latest values. DZiG [16] develops a DelZero-Aware incremental refinement strategy to efficiently handle computation sparsity during incremental processing. The intermediate state in both, GraphBolt and DZiG, requires much larger amount of memory compared to KickStarter, which drastically increases their memory footprint. Our memory-efficient stateful iterative models limit the memory footprint by reducing the amount of intermediate state that gets captured for efficient incremental computation.

GraphInc [4] saves all messages across edges along with computed states in order to replay the computation with incremental changes. This ends up demanding much larger memory capacity (intermediate state in the order of edges for each iteration) compared to recent works like GraphBolt. Differential Dataflow [18] is a general-purpose system that enables incremental computation by tracking states at operator level, i.e., for graph computations it also maintains intermediate state in the order of edges per iteration, which results in a much larger memory footprint than KickStarter and GraphBolt. Since our memory-efficient stateful iterative models prune out intermediate state at vertex-level, they can be easily applied to these systems by selectively tracking intermediate edge-level or operator-level states for only a subset of vertices.

Systems like [5, 6, 7, 13, 15, 27, 28, 30] use stateless iterative models which limits their efficiency in delivering accurate results upon graph mutation. Tornado [28], Kineograph [5] and GraphIn [27] perform incremental computation by triggering the user functions based on graph updates and allowing the changes to propagate throughout the graph. Hence, they can-

not guarantee accurate results for BSP algorithms. GraphIn identifies the vertices that could be potentially impacted by graph updates using tag propagation, and restarts computation from scratch for those identified vertices. [30] uses GIM-V (generalized iterative matrix vector multiplication) to perform incremental computation. LLAMA [15], STINGER [7], Aspen [6], GraphOne [13] and LiveGraph [43] focus on designing efficient dynamic graph data structures and storage systems, and their processing models do not support incremental computation. However, incremental computation is crucial in delivering high end-to-end performance, as shown by detailed performance comparison of these systems in [16].

Solutions like [10, 11, 19, 34] operate on evolving graphs that contain a group of temporally-related graph snapshots capturing the evolution of the graph structure over time. These systems do not capture intermediate states and perform incremental computation by directly reusing the results computed for previous graph snapshots, making them suitable for self-fixing graph algorithms but not for general BSP algorithms.

Finally, static graph processing systems [8, 9, 14, 20, 21, 24, 25, 26, 29, 33, 36, 38, 42] focus on processing a given static graph input and do not natively handle dynamic graphs with incremental processing.

7 Conclusion

We presented two memory-efficient stateful iterative models for incremental processing of streaming graphs. The Selective Stateful Iterative Model controls the memory footprint by selecting a small portion of the intermediate state to be maintained throughout execution. The Minimal Stateful Iterative Model further reduces the memory footprint by exploiting the distributive update property in graph algorithms. Our results showed that our models are effective in limiting the memory footprint while still retaining most of the performance benefits of traditional stateful iterative models, hence being able to scale on larger graphs that could not be handled by the traditional models.

8 Acknowledgements

We would like to thank our shepherd Vasiliki Kalavri and the anonymous reviewers for their valuable feedback. This work is supported by Oracle for Research and the Natural Sciences and Engineering Research Council of Canada.

References

- [1] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered Label Propagation: A Multi-Resolution Coordinate-Free Ordering for Compressing Social Networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the International Conference on World Wide Web (WWW '11)*, pages 587–596, 2011.
- [2] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A Large Time-Aware Web Graph. In *Special Interest Group on Information Retrieval (SIGIR '08)*, pages 33–38, 2008.
- [3] Paolo Boldi and Sebastiano Vigna. The WebGraph Framework I: Compression Techniques. In *Proceedings of the International World Wide Web Conference (WWW '04)*, pages 595–601, 2004.
- [4] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. Facilitating Real-Time Graph Mining. In *Proceedings of the International Workshop on Cloud Data Management (CloudDB '12)*, pages 1–8, 2012.
- [5] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the Pulse of a Fast-Changing and Connected World. In *Proceedings of the European Conference on Computer Systems (EuroSys '12)*, pages 85–98, 2012.
- [6] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-Latency Graph Streaming Using Compressed Purely-Functional Trees. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*, pages 918–934, 2019.
- [7] David Ediger, Rob McColl, Jason Riedy, and David A Bader. Stinger: High Performance Data Structure for Streaming Graphs. In *IEEE Conference on High Performance Extreme Computing (HPEC '12)*, pages 1–5, 2012.
- [8] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed Graph-Parallel Computation on Natural Graphs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 17–30, 2012.
- [9] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 599–613, 2014.
- [10] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: A Graph Engine for Temporal Graph Analysis. In *Proceedings of the European Conference on Computer Systems (EuroSys '14)*, pages 1–14, 2014.
- [11] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-Evolving Graph Processing at Scale. In *Proceedings of the International Workshop on Graph Data Management Experiences and Systems (GRADES '16)*, pages 1–6, 2016.
- [12] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. CuSha: Vertex-Centric Graph Processing on GPUs. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC '14)*, pages 239–252, 2014.
- [13] Pradeep Kumar and H Howie Huang. Graphone: A Data Store for Real-Time Analytics on Evolving Graphs. In *USENIX Conference on File and Storage Technologies (FAST '19)*, pages 249–263, 2019.
- [14] Heng Lin, Xiaowei Zhu, Bowen Yu, Xiongchao Tang, Wei Xue, Wenguang Chen, Lufei Zhang, Torsten Hoefler, Xiaosong Ma, Xin Liu, Weimin Zheng, and Jingfang Xu. ShenTu: Processing Multi-Trillion Edge Graphs on Millions of Cores in Seconds. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*, pages 706–716, 2018.
- [15] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *IEEE International Conference on Data Engineering (ICDE '15)*, pages 363–374, 2015.
- [16] Mugilan Mariappan, Joanna Che, and Keval Vora. DZiG: Sparsity-Aware Incremental Processing of Streaming Graphs. In *Proceedings of the European Conference on Computer Systems (EuroSys '21)*, pages 1–16, 2021.
- [17] Mugilan Mariappan and Keval Vora. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the European Conference on Computer Systems (EuroSys '19)*, pages 1–16, 2019.
- [18] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential Dataflow. In *Conference on Innovative Data Systems Research (CIDR '13)*, 2013.
- [19] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. ImmortalGraph: A System for Storage and Analysis of Temporal Graphs. *ACM Transactions on Storage (TOS '15)*, 11(3):1–34, 2015.

- [20] Svilen R. Mihaylov, Zachary G. Ives, and Sudipto Guha. REX: Recursive, Delta-Based Data-Centric Computation. *Proceedings of the VLDB Endowment (PVLDB '12)*, 2012.
- [21] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Symposium on Operating Systems Principles (SOSP '13)*, pages 456–471, 2013.
- [22] Kamal Nigam and Rayid Ghani. Analyzing the Effectiveness and Applicability of Co-training. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM '00)*, pages 86–93, 2000.
- [23] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford InfoLab, 1999.
- [24] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the Symposium on Operating Systems Principles (SOSP '15)*, page 410–424, 2015.
- [25] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-Centric Graph Processing Using Streaming Partitions. In *Proceedings of the Symposium on Operating Systems Principles (SOSP '13)*, page 472–488, 2013.
- [26] Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM '13)*, pages 1–12, 2013.
- [27] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. Graphin: An Online High Performance Incremental Graph Processing Framework. In *European Conference on Parallel Processing (Euro-Par '16)*, pages 319–333, 2016.
- [28] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. Tornado: A System For Real-Time Iterative Analysis Over Evolving Data. In *Proceedings of the International Conference on Management of Data (SIGMOD '16)*, pages 417–430, 2016.
- [29] Julian Shun and Guy E Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*, pages 135–146, 2013.
- [30] Toyotaro Suzumura, Shunsuke Nishii, and Masaru Ganse. Towards Large-scale Graph Stream Processing Platform. In *Proceedings of the International Conference on World Wide Web (WWW '14)*, pages 1321–1326, 2014.
- [31] Hanghang Tong, Jingrui He, Mingjing Li, Changshui Zhang, and Wei-Ying Ma. Graph based multi-modality learning. In *Proceedings of the International Conference on Multimedia (MULTIMEDIA '05)*, pages 862–871, 2005.
- [32] Leslie G Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, pages 103–111, 1990.
- [33] Keval Vora. Lumos: Dependency-Driven Disk-based Graph Processing. In *USENIX Annual Technical Conference (USENIX ATC '19)*, pages 429–442, 2019.
- [34] Keval Vora, Rajiv Gupta, and Guoqing Xu. Synergistic Analysis of Evolving Graphs. *ACM Transactions on Architecture and Code Optimization (TACO '16)*, pages 1–27, 2016.
- [35] Keval Vora, Rajiv Gupta, and Guoqing Xu. Kickstarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, pages 237–251, 2017.
- [36] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms Using a Relaxed Consistency Based DSM. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*, page 861–878, 2014.
- [37] Yang Wang, Muhammad Aamir Cheema, Xuemin Lin, and Qing Zhang. Multi-Manifold Ranking: Using Multiple Features for Better Image Retrieval. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD '13)*, pages 449–460, 2013.
- [38] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. GraM: Scaling Graph Computation to the Trillions. In *Proceedings of the Symposium on Cloud Computing (SoCC '15)*, pages 408–421, 2015.
- [39] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Maiter: An Asynchronous Graph Processing Framework for Delta-based Accumulative Iterative Computation. *IEEE Transactions on Parallel and Distributed Systems (TPDS '13)*, 25(8):2091–2100, 2013.

- [40] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *International Conference on Algorithmic Applications in Management (AAIM '08)*, pages 337–348, 2008.
- [41] Xiaojin Zhu and Zoubin Ghahramani. Learning From Labeled and Unlabeled Data with Label Propagation. *Tech. Rep., Technical Report CMU-CALD-02-107, Carnegie Mellon University*, 2002.
- [42] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 301–316, 2016.
- [43] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proceedings of the VLDB Endowment (PVLDB '20)*, page 1020–1034, 2020.

Avocado: A Secure In-Memory Distributed Storage System

Maurice Bailleu¹, Dimitra Giantsidi¹

Vasilis Gavrielatos¹, Do Le Quoc^{2*}, Vijay Nagarajan¹, Pramod Bhatotia^{1,3}

¹University of Edinburgh ²Huawei Research ³TU Munich

Abstract

We introduce AVOCADO, a secure in-memory distributed storage system that provides strong security, fault-tolerance, consistency (linearizability) and performance for untrusted cloud environments. AVOCADO achieves these properties based on TEEs, which, however, are primarily designed for securing limited physical memory (enclave) within a single-node system. AVOCADO overcomes this limitation by extending the trust of a secure single-node enclave to the distributed environment over an untrusted network, while ensuring that replicas are kept consistent and fault-tolerant in a malicious environment.

To achieve these goals, we design and implement AVOCADO underpinning on the cross-layer contributions involving the network stack, the replication protocol, scalable trust establishment, and memory management. AVOCADO is practical: In comparison to BFT, AVOCADO provides confidentiality with fewer replicas and is significantly faster—4.5× to 65× for YCSB read and write heavy workloads, respectively.

1 Introduction

In-memory distributed key-value stores (KVS) [8, 29–31, 40, 44, 45, 61, 66, 88, 106] have been widely adopted as the underlying storage system infrastructure in the cloud because (i) they support latency sensitive applications by keeping data in main memory, and (ii) they are able to accommodate large datasets beyond the memory limits of a single server by adopting a scale-out distributed design.

At the same time, the transition to the cloud has increased the risk of security violations in storage systems [77]. In untrusted environments, an attacker can compromise the security properties of the stored data and query operations. In fact, several studies [36, 37, 83] show that software bugs, configuration errors, and security vulnerabilities pose a serious threat to storage systems. Further, a malicious cloud operator or co-located tenant, presents an additional attack vector [78, 79].

To address these security threats, hardware-assisted trusted execution environments (TEEs), such as Intel SGX [5], ARM Trustzone [12], RISC-V Keystone [53, 76], and AMD-SEV [10] provide an appealing way to build secure systems. In particular, TEEs provide a hardware-protected secure memory region whose residing code and data are isolated from any layers in the software stack including the OS/

hypervisor. Given this promise, TEEs are now commercially offered by major cloud computing providers [23, 34, 60].

Although TEEs provide a promising building block for securing systems against a powerful adversary, they also present significant challenges while designing a *replicated secure distributed storage system*. The fundamental issue is that the TEEs are primarily designed to secure the limited in-memory state of a single-node system, and thus, the security properties of TEEs do not naturally extend to a distributed infrastructure. Therefore we ask the question: *How can we leverage TEEs to design a high-performance, secure, and fault-tolerant in-memory distributed KVS for untrusted cloud environments?*

In this work we introduce AVOCADO, a secure, distributed in-memory KVS based on Intel SGX [5] as the foundational TEE that achieves the following properties: (a) strong *security*, in particular, *confidentiality*—unauthorized reads are prevented, and *integrity*—unauthorized changes to the data are detected, (b) *fault tolerance*—the service continues uninterrupted in the presence of faults, (c) *consistency*—strong consistency semantics for a replicated store (linearizability), while protecting against roll back and forking attacks and (d) *performance*—achieving all of these without compromising performance.

To achieve the aforementioned properties, we need to address the following four design challenges pertaining to the network stack, the replication protocol, trust establishment, and memory management in TEEs.

Firstly, in-memory distributed KVSs are increasingly build on high-performance network stacks, where they bypass the kernel using direct I/O [4, 30, 46]. Unfortunately, the prominent I/O mechanism employed by TEE frameworks [13, 65, 72] is based on asynchronous system calls [85], which exhibit significant overheads [104]. On the other hand, the direct I/O mechanism is fundamentally incompatible with TEEs as the data stored within the protected memory of TEEs cannot be directly accessed via the untrusted DMA connection.

To address this challenge, we design a high-performance network stack for TEEs based on eRPC [46]—it supports the complete transport and session layers, while enabling direct I/O within the protected TEE domain. Our network stack outperforms asynchronous syscall by 66% for iperf (§6.1).

Secondly, in-memory distributed KVSs rely on data replication for fault tolerance. To ensure replicas are consistent in the presence of faults and adversary, a secure replication protocol is deployed. While conventional wisdom requires the employment of BFT protocols [20, 52], they are prohibitively

*Do Le Quoc performed this work at TU Dresden.

expensive for practical systems [22].

To overcome the limitation, we design a secure replication protocol, which builds on top of any high-performance non-Byzantine protocol [56, 98]—our key insight is to leverage TEEs to preserve the integrity of protocol execution, which allows to model Byzantine behavior as a normal crash fault. Our replication protocol offers linearizable reads and writes, and outperforms BFT [87] by a factor of $4.5\times$ – $65\times$, while requiring f fewer replicas and stronger security properties (§ 6.2).

Thirdly, a secure distributed system requires a scalable attestation mechanism to establish trust between the servers and clients. Unfortunately, the remote attestation mechanism in TEEs is designed for establishing root of trust for a single node [68] and it does not provide collective trust establishment across the multiple nodes of a distributed system. Moreover, the attestation itself is based on Intel attestation service (IAS) [3, 11], which suffers from scalability and latency issues.

To address this, we design a configuration and attestation service (CAS) that ensures scalability and flexibility in a distributed environment. Further, it provides configuration management, and improved performance of $18.3\times$ compared to Intel’s IAS attestation (§ 6.3).

And lastly, an in-memory distributed KVS requires fast access to large amount of main memory on each server for single-node KVS. Unfortunately, TEEs provide a limited secure physical memory, and rely on prohibitively expensive paging mechanism to access data beyond the physical limit.

To address this limitation, we design a novel single-node KVS based on a partitioned skip list data structure, which overcomes the memory limitations of TEEs, while supporting lock-free scalable concurrent updates. Our KVS provides fast lookup speed; $1.5\times$ – $9\times$ faster than ShieldStore [48], a state-of-the-art secure KVS for single-node systems (§ 6.4).

Based on these aforementioned four contributions, we build AVOCADO as an end-to-end system from the ground-up, and evaluate it using a real hardware cluster using the YCSB [7, 24] workloads. Our evaluation shows that AVOCADO is scalable and performs similar in read heavy and write heavy workloads: AVOCADO suffers only 50 % slowdown compared to a non-secure distributed KVS (§ 6.5), which is an order of magnitude better than the state-of-the-art secure distributed storage systems providing strong consistency (§ 7).

Limitations: AVOCADO requires a large trusted computing base (TCB) compared to other work using TEE to provided secure replication [18, 25, 26]. While BFT protocols can handle implementation errors, AVOCADO cannot and requires the TCB to be implemented correctly. Further, we do not aim to protect against side-channel attacks and access or network pattern attacks [37, 49, 55, 105]. Protecting against these attacks is outside of the scope of this work.

2 Background

2.1 Trusted computing

Trusted Execution Environments (TEEs) [5, 10, 12, 27, 76] are tamper-resistant processing environments that guarantee the authenticity, the integrity and the confidentiality of their executing code, data and runtime states, e.g. CPU registers, memory and others. Their content remains resistant against all software attacks even from privileged code (OS, hypervisor).

SGX, Intel’s version of a TEE, offers the abstraction of an isolated memory called *enclave*. Enclave pages reside in the enclave page cache (EPC) — a specific memory region (up to 128 MiB for v1 and 256 MiB for v2) which is protected by an on-chip memory encryption engine (MEE). To support applications with larger memory footprint SGX implements a *paging* mechanism. However, the EPC paging mechanism incurs high overheads [16, 65].

This isolation prohibits SGX applications from executing outside-of-the-enclave code directly. Thus, enclave threads need to *exit* the trusted environment and further copy all associated data out of the enclave since kernel code cannot access it. Afterwards, threads have to *enter* the enclave again. We refer to this as *world switch*.

Our AVOCADO project leverages the advancements in shielded execution frameworks; in particular, we use SCONE [13] to build a distributed storage system.

2.2 High-performance networking with eRPC

Traditionally the network stack and the I/O are handled inside the OS kernel where conventional applications perform *system calls* to send/receive messages. However, context switches due to system calls present a bottleneck and might sacrifice performance [2, 38, 43, 86, 101]. Consequently, approaches like RDMA and DPDK [4] are widely favored in high performance networking because they (i) can map a device into the users address space, and (ii) replace the costly context switches with a polling-based approach.

In our work, we build our network stack based on eRPC [46], a state-of-the art general-purpose and asynchronous remote procedure call (RPC) library for high-speed networking for lossy Ethernet or lossless fabrics. eRPC uses a polling-based network I/O along with userspace drivers, eliminating interrupts and system call overheads from the datapath.

Lastly, eRPC provides us with a UDP stack, leverages optimization techniques (e.g. zero-copy reception, congestion control, etc.) while it remains generic; it supports a wide range of transport layers such as RDMA, DPDK, and RoCE.

3 System model

AVOCADO divides the key space into shards. Each shard is replicated over a configurable number of nodes, which are connected over a high speed network. A client issuing a Put, Get, or Delete operation selects the shard associated with the key and chooses a *request coordinator* from the list of nodes. The nodes will coordinate with each other

to provide proof for the success of the operation. For Get operations proofs of integrity, authenticity, consistency and non-/existence need to be provided, too.

Data model. AVOCADO provides confidentiality, integrity, authenticity and strong consistency for the stored data. Specifically, a server only acknowledges a request as long as it can prove the following guarantees: 1) an adversary cannot read or manipulate stored data, without the manipulation being detected, 2) the servers can establish trust with each other and the clients and 3) an operation always observes the latest completed operation on the same key e.g., a Get observes the latest Put.

Threat model. AVOCADO targets an extended threat model beyond the conventional model assumed for single-node shielded execution [17]. In line with the default threat model of SGX, we assume that an adversary has full control over the hardware and software stack of the provided system, including OS and hypervisor. Further, the adversary has the ability to gain full control over the network infrastructure and can drop, delay, or manipulate network traffic. In contrast to BFT protocols, we assume that adversary cannot take advantage of faults in the implementation of SGX or KVS. Moreover, our work does not protect against side-channel attacks [42, 49, 55, 62, 81, 97, 99, 100]. AVOCADO also does not provide mechanisms against access pattern attacks [37, 105]. Lastly, we also do not protect against memory safety vulnerabilities in our implementation [51, 63].

Fault model. We assume an asynchronous model with network and crash-stop failures. The network can be manipulated by the attacker, thus, we assume that message transmission delays can be unbounded, network packets can be reordered, lost or duplicated. We do not assume the existence of synchronized clocks. Individual processes might fail by crashing, but do not operate in a Byzantine manner (because of trusted execution in the nodes). Since the network is controlled by the attacker, AVOCADO cannot provide any availability guarantees. However, as long as there is not a denial-of-service attack on the network, AVOCADO will remain available while a majority of processes remain alive (tolerating f failures).

4 Design

AVOCADO, as a distributed KVS, runs on a set of nodes, each of which has to continuously guarantee the confidentiality, integrity and authenticity of the stored data as well as the sent/received messages. As shown in Figure 1, each node consists of four major components. On the top, a configuration and attestation service (CAS) runs to provide and speed up the trust establishment between the nodes and the clients. Additionally, AVOCADO guarantees fault tolerance as well as consistency between the replicated nodes thanks to an asynchronous replication protocol. We implement this replication protocol using our secure network stack. Further, the network stack securely sends and receives messages, ensuring packet security. Finally, the single-node memory

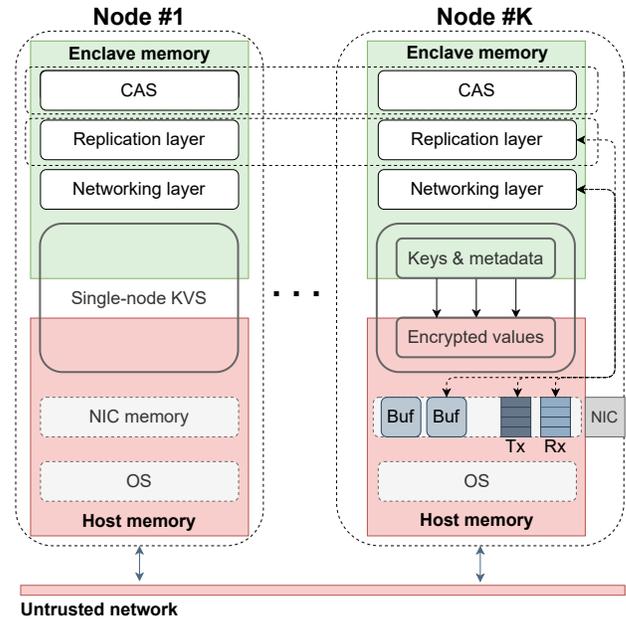


Figure 1: System overview.

KVS stores the dataset, containing all user provided data and ABD’s timestamps.

Following, we will discuss the four major components, i.e. network stack, replication protocol, CAS, and in-memory KVS, in more detail.

4.1 Network stack

Problem. High-performance networking based on direct I/O mechanisms (e.g., RDMA and DPDK) is an essential ingredient to design a distributed in-memory KVS. The networking layer is imperative to support high-performance synchronous replication. Unfortunately, mapping devices into TEEs trusted memory is incompatible to the security guarantees, since the device is untrusted. Direct I/O mechanisms, however, depend on DMA.

Additionally, the synchronous socket syscall I/O is limiting as it requires the expensive world switch in the TEEs (the world switch is around 5.5× more expensive than a kernel context switch; 10,170 cycles compared to 1,800 cycles [104]).

To prevent the expensive world switches, asynchronous syscall mechanisms [85] have been adopted by shielded execution frameworks, such as SCONE [13] or Eleos [65]. Although the asynchronous syscall mechanism helps in mitigating the expensive world switch in TEEs, it is not well-suited for AVOCADO since the system call overhead as well as copying data in/out the enclave memory are not avoided. For example, in our evaluation in Section 6.1 we prove that the exit-less asynchronous socket-based networking is poor choice compared to a userspace approach for AVOCADO.

Solution. To overcome this limitation, we opted for a new network stack based on the userspace direct I/O networking approaches (e.g., RDMA and DPDK), offering a secure implementation of the transport and session layer in the OSI

model. However, we need to tackle the fact that untrusted resources/memory cannot be mapped into the enclave memory. To address this, our network stack maps the DMA, and message buffers into the untrusted host memory, which is accessible by the enclave.

Shielded network stack. For our shielded network stack, we use eRPC [46] on top of DPDK [4]. To strengthen AVOCADO’s security properties and eliminate world switches, we also map all eRPC’s and DPDK’s software stack to the enclave address space by leveraging Scone. Therefore, the logic, i.e. code, *lives* completely within the enclave while the networking buffers (e.g. message buffers, network protocol buffers, Tx and Rx queues) remain in host memory since SGX will not allow registering enclave memory to the NIC. As a result, we map untrusted host memory to both NIC and network buffers required by eRPC and we utilise hugepages memory of 2 MB-pages to boost packet processing (e.g. eliminate page walks, exploit data locality, minimize swapping, and increase TLB hit rate).

As shown in Figure 2, the submission and reception of requests and responses mandate the allocation of message buffers. To transmit the message buffer’s data, eRPC needs to copy the data to a `rte_mbufs` in the Tx queue which is allocated by DPDK library and also resides in hugepages area. However, before that, a header that contains the transport header, and metadata (request handler type, sequence numbers, etc.) is added to the front of the packet. Specifically, eRPC library adds the UDP protocol header while the DPDK library is responsible for the Ethernet protocol header.

Upon a request’s reception, a specific handler for the type of the request is invoked. The Rx queue’s elements are pointers to the address of the received data. In case the packet is smaller than the MTU (1500 B in our case), we perform zero-copy reception by mapping the data address to the message buffer associated for that request. Our networking stack splits big packages (> MTU) into a set of ordered MTU-size smaller messages and delivers them in order—we guarantee to order by unique monotonic sequence ids. The first (sub)-message contains all the necessary metadata (e.g. the size of the original message). That way, if a message is lost, our library identifies the missing part and only the lost message is re-transmitted. Lastly, note that each user thread owns a separate RPC object which owns distinct Tx and Rx queues allowing that way multithreaded concurrent operations.

Encryption and message format. To sum up, AVOCADO efficiently eliminates the world switches, establishes a direct communication with the device bypassing the kernel network stack, attacks the limited enclave memory and promotes parallelism. However, by putting these buffers outside the hardware protected area, AVOCADO has to ensure the integrity and confidentiality for all network data. Towards this direction, we implemented an en-/decryption library (using hardware support for AES-GCM-128). Each call or return from eRPC goes through this en-/decryption layer which also checks the integrity of the transmitted data.

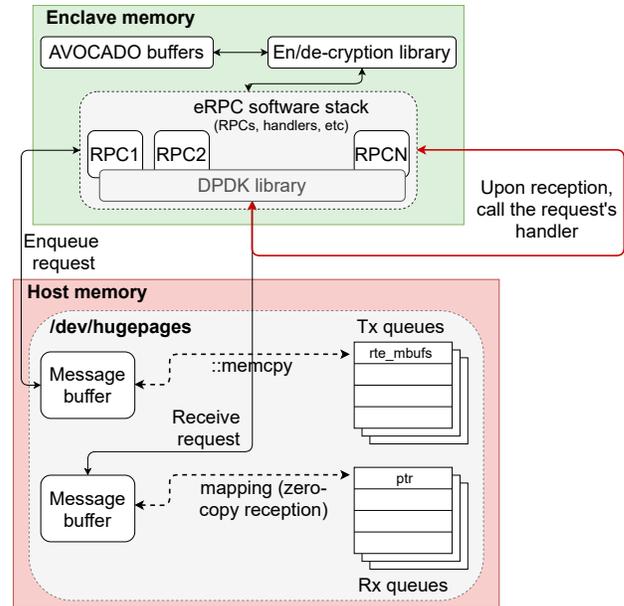


Figure 2: AVOCADO’s network stack.

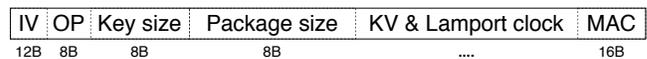


Figure 3: AVOCADO’s message format.

Figure 3 shows the message format of our AVOCADO-networking layer. For each transmitted packet, the encryption layer builds a payload, which contains a 12 B IV, 8 B operation identifier, 8 B for the key size, 8 B for the size of the entire package then the KV pair with the Lamport clock (§ 4.2). The generated payload is followed by a 16 B MAC which is necessary to prove the authenticity and integrity upon reception in the remote host. The operation identifier also contains a unique id for the current request/response, allowing to detect resend packages by an attacker. Replicas are trusted and all replicas authenticate each other in the boot up step. Further, they make a key exchange, therefore we can use this together with the unique id to authenticate each message. By encrypting and authenticating our packages we can deal with security concerns raised for user space networking [84, 94].

Result. In Section 6.1 we show that our userspace shielded network stack based on eRPC outperforms the kernel approach based on sockets up to 1.66x.

4.2 Replication protocol

Problem. Distributed systems enforce consistency in the face of faults through replication protocols that establish an order of operations in a replicated environment, preventing data corruption and loss. We strive for linearizability [39], the strongest guarantee from a programmability perspective, which mandates that each request appears to take effect globally and instantaneously at some point between its invocation and completion. Additionally, we strive to provide two often contradictory properties: security and

Protocols	Linearizability	Integrity	Confidentiality	Replication factor	Max compromised nodes
Non-Byzantine	✗	✗	✗	$2f+1$	0
BFT	✓	✓	✗	$3f+1$	f
BFT + TEEs	✓	✓	✓	$3f+1$	All
AVOCADO	✓	✓	✓	$2f+1$	All

Table 1: The landscape of replication protocols in the untrusted environment. This table compares theoretical systems with different protocols against AVOCADO. Thereby, all the systems utilize a secure single-node KVS, however only the execution of BFT + TEE is protected. We assume f compromised nodes for linearizability, integrity and confidentiality columns.

performance. Conventional wisdom suggests the use of BFT protocols [20, 52] since they provide a secure consensus protocol in a malicious environment. However, their performance suffers from their overly pessimistic assumptions. On the other hand, non-Byzantine replication protocols, such as ABD [14, 56], chain replication [98] or Raft [64], perform better than BFT, but cannot tolerate a malicious environment.

Solution. Since AVOCADO assumes a malicious environment, BFT [20, 52] protocols could be deployed to deal with malicious responses. Prior work uses trusted components to increase the performance of BFT protocols by detecting equivocation [18, 54]. However, our assumption of the system differs from BFT. In contrast to BFT, we assume that enclaves will respond correctly, preventing equivocation. Furthermore the TEE is able to preserve the integrity of the protocol execution. This allows us to model a Byzantine behavior as a normal crash fault. As a result, we can adopt a non-BFT replication protocol, which deals with crash faults. Thereby, our design greatly increases the performance by avoiding the additional broadcast rounds required by BFT, while also reducing the required nodes to tolerate f failures. In Table 1 we compare security guarantees of different protocols with and without TEEs.

In AVOCADO we build our replication protocol on the well established multi-writer ABD [56] protocol. (From now on “ABD”.) By choosing ABD, we can also guarantee protection against forking and rollback attacks. ABD requires a majority of nodes to acknowledge each operation, guaranteeing that at least one replica involved in the operation has observed the most recent operation on the same key. This further guarantees liveness in case of network partitioning as long as a majority of nodes are in the same partition. While we do not change the replication-related behavior of the original ABD protocol, we design a secure replication protocol based on our network stack (§ 4.1). In the following we describe the important operations of AVOCADO.

#I: Put. In a Put operation the client will determine, by hashing the key and looking up the nodes, the set of nodes responsible for the key. They, then, send the Put to a randomly selected replica, which will act as the Put’s coordinator.

The chosen request’s coordinator will prepare it’s own KVS by preparing the local put operation, however it will not make the local put visible for other operations until the replicated Put operation is completed. This reduces EPC pressure, since the value doesn’t have to be cached in enclave memory before

it can be inserted into the nodes KVS. An example of the AVOCADO’s Put request is shown in Figure 4. The coordinator, first, executes the first of two broadcast rounds. All replicas store the key-value along with its Lamport clock to determine an order of operations. The Lamport clock consists of a logical counter and a machine id. This id guarantees that only one machine can generate a specific clock value. In the first broadcast round, the coordinator requests the timestamps that are stored in the replicas for that key. All replicas lookup the key in their in-memory KVS, to find their stored timestamp. Crucially, the replicas do not have to make an authenticity and integrity check on the timestamp, as the Lamport clock is stored as part of the metadata in enclave memory. Upon receiving a majority of the remote timestamps (including its own locally stored timestamp), the coordinator creates the timestamp of the new Put, by incrementing the highest of the received timestamps and concatenating its own node-id. Finally, it broadcasts the new KV pair along with its new timestamp to all replicas, which insert the KV pair into their in-memory KVS. Since the put operation does not return the value to the user, and the meta data is protected by the enclave AVOCADO does not have to check the authenticity and integrity of the old value. Upon gathering a majority of acknowledgements it reports completion to the client.

#II: Get. The Get operation is similar to Put; the client sends its request to a randomly selected server, which coordinates it. The server then looks up the KV-pair in its local store.

The chosen request coordinator executes one broadcast round. In certain cases a second, optional broadcast round is required. Similarly, to the first round of a Put, the first round of a Get finds out the highest timestamp for that key when the majority of replicas has responded. This action guarantees that the Get will observe any completed Put (recall that a Put only completes if it reaches a majority of replicas). The replicas will respond with their locally stored value and corresponding Lamport timestamp to the coordinator, this involves a lookup in the local KVS and decryption together with integrity and authenticity checks of the value. The Get always returns to the client the value that corresponds to the highest timestamp found in its first round. However, the coordinator can reply to the client iff, based on the replies it received on its first round, it can guarantee that a majority of replicas are aware of this value. Otherwise it must perform a second broadcast round.

The second broadcast round is identical to the second write of a Put: it shares the KV-pair along with its timestamp

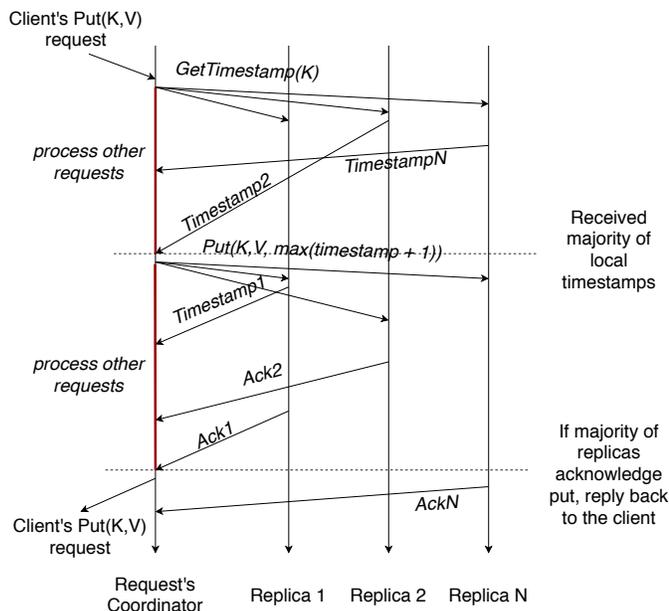


Figure 4: Example of Put request in AVOCADO protocol.

with all replicas. Completion of the Get is reported to the client only after gathering a majority of acks. The second round of the Get, ensures that a Get not only observes the latest completed Put, but also guaranteeing that the Put will be visible to all subsequent Gets.

#III Delete. Delete is supported by issuing a Put operation with an empty value. This will remove the value from the KVS, but importantly it will not remove the key. We need to keep the key and the corresponding Lamport clock, to be able to establish an order of operation if a future Put operation accesses the same key.

#IV: Fault tolerance. AVOCADO remains highly available in the face of machine failures. However, as nodes fail, new nodes must be added, to ensure that the deployment always includes a majority of live nodes. In order to ensure that machines can safely join the configuration, we deploy a recovery algorithm inspired by Hermes [47].

Specifically, when adding a new node all other live replicas are notified of the new node’s intention to join the replica group. The new node starts operating as a *shadow replica* that participates in all Put-related broadcast rounds (of remote replicas), but it cannot yet become the coordinator of a client request. Furthermore, the shadow replica does not take part in the Get quorum. In the meantime, the shadow replica reads chunks (multiple keys) from other replicas to fetch the latest values and reconstruct the KVS. To archive this the shadow replica is using the first broadcast round of ABD, but it never executes the second round, because it does not need to notify other replicas of what it read. After reading the entire KVS, the shadow replica is up-to-date and transitions to operational state, whereby it is able to serve client requests.

Result. We compare AVOCADO against BFT and Raft in

Section 6.2. Our evaluation shows that AVOCADO is between 4.5 and 65× faster than BFT-Smart [87].

4.3 Configuration and attestation service

Problem. To ensure the integrity of the code and data deployed in the remote hosts with TEEs, TEEs, such as Intel SGX, provide attestation mechanisms. Secrets (e.g. certificates, encryption keys, etc.) are provided only after the attestation. Once an enclave is initialized, an attestation process can be launched to verify the integrity of code and data inside the enclave and proves the enclaves identity to a remote party.

Intel SGX uses an architecture Platform Service Enclave (PSE) called *Quoting Enclave* to sign the report of the loaded enclave [11, 27]. The remote verifier forwards this signed report to the Intel Attestation Service (IAS). Thereafter, IAS confirms or refuses the authenticity of the report to the verifier.

This conventional attestation mechanism using IAS incurs significant overhead in a distributed setting, especially for elastic computing or fault tolerance. The reason is that every time a distributed system (e.g. a distributed KVS) spawns a new enclave, it needs to perform the remote attestation via IAS which is not necessarily hosted in the same data center, incurring high latency. Lastly, and importantly, cloud providers usually do not want to disclosure their hardware or cluster information, as this information might be confidential.

Solution. In AVOCADO, we overcome this challenge by designing a decentralized configuration and attestation management system (CAS) for distributed SGX-based applications.

By consolidating and expanding the traditional attestation mechanism of Intel to build our CAS, we automatically and transparently perform the attestation for each node. The key idea behind our design is that we replace the Quoting Enclave in the Intel attestation mechanism by the LAS. The CAS first attests the LAS using the Intel attestation mechanism, thereafter the LAS will operate as the root of trust in our remote attestation mechanism. Note, that we can launch as many LAS instances as required for availability. The LAS performs the local attestation for AVOCADO nodes and provides attestation quotes that can be verified by the CAS. Thus, our mechanism does not need to interact with IAS after the LAS is trusted, this reduces significantly the overhead of the traditional attestation. We achieve the *transparent and automatic properties* by deeply embedding the remote attestation into the AVOCADO runtime. In addition, our CAS only provisions a configuration and secrets to execute AVOCADO once it ensures that all nodes were not manipulated. Each node of AVOCADO can only communicate with others if it can provide a valid certificate provided by our CAS. Therefore, users can just rely on the CAS to control and operate other components of AVOCADO. They only have to attest our CAS before providing secrets to it. The CAS itself also runs inside an enclave, thus users can use the traditional attestation method of Intel to validate it.

Result. As shown in 6.3 our CAS achieves 18.2× lower end-to-end latency in AVOCADO when comparing with IAS.

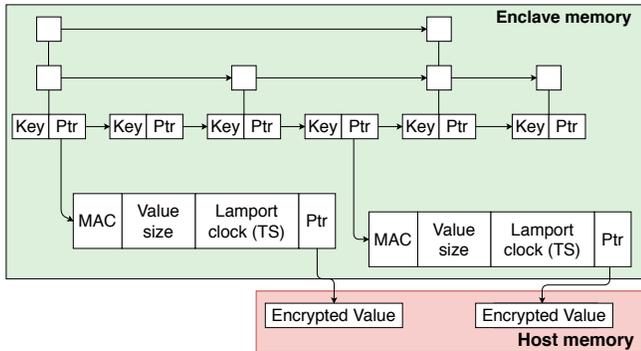


Figure 5: AVOCADO’s single-node KVS.

4.4 Single-node KVS

Problem. Enclave’s memory limitation is in stark contrast to the requirements of designing an in-memory KVS, which requires fast access to large amounts of in-memory data. Unfortunately, enclaves provide only limited physical memory (94 MiB) and incur high overheads due to the EPC paging mechanism (2–2000× [51]) beyond it.

To overcome the limitations of the strawman design, ShieldStore [48], a state-of-the-art secure in-memory KVS for a single-node system, proposed a MerkleTree-like data structure design where the entire KVS resides in the untrusted host memory, except for the security metadata (hash buckets heads). The metadata stored in the enclave memory is used to speed up the look up and to perform authenticity and integrity checks on the KV pairs. However, in our experience, the ShieldStore design suffers from continuous integrity re-calculations. Furthermore, the memory layout prohibits efficient concurrent operations.

Compared to Speicher [16], which also introduced the KV pair separation scheme for enclaves, our KVS is optimized for paging by encountering locality.

Solution. To overcome these limitations, we designed and propose our own in-memory concurrent data structure for the single-node KVS. Our KVS is based on a authenticated and confidentiality-preserving skip list [74] which supports secure and fast updates and lookups. We have chosen skip list as our fundamental data structure because it maintains the best features of a sorted array for searching ($O(\log n)$) and of a linked list-like structure for insertion ($O(\log n)$). Our design carefully partitions the key and value space by placing the keys along with metadata inside the enclave memory, while storing the bulk of data encrypted and integrity and authenticity protected in the untrusted host memory. Our partitioned data structure (keys and values) allow for faster lookups than ShieldStore’s hash buckets, while it also reduces the amount of necessary calculations to guarantee the integrity and authenticity. Furthermore, our lock-free data structure supports concurrent operations and it is well-suited for increased parallelism.

As shown in Figure 5 the nodes of the skip list reside

inside the enclave and contain the key and a pointer to metadata structure. This structure contains the 16 B MAC, for guaranteeing the integrity and authenticity of the value. Furthermore, the data structure also includes the size of the value, which makes checks on the value easier, since we do not need to read any information from the untrusted host memory, to retrieve how many bytes should be read. AVOCADO’s consistency protocol uses logical clocks, i.e. Lamport clocks, to establish an order of operations on each key (§ 4.2). Therefore, we also store the Lamport clock in the corresponding metadata block, to prevent costly decryptions on the timestamp queries. Lastly, the metadata structure also stores the pointer to the value in the untrusted host memory.

Importantly, separating the metadata and the bulk data (i.e. values) from the skip list allows us to update the skip list *lock free*. Further, it also decreases the EPC pressure when doing a lookup, as nodes can be stored more compact and the metadata can be stored on a different page. However, looking up a value mandates an additional indirection due to keys and metadata separation. Nevertheless, we believe that updating the KVS without acquiring any locks is worth this additional indirection as it allows better multi-threaded scalability. Therefore, in contrast to a HashBucket design like ShieldStore, we never need to stall. In contrast to ShieldStore our approach seems to be limited by the enclave memory, however, assuming we have 1 KiB values and 16 B keys, we achieve a space reduction for enclave memory of 92.8 % compared to a naive implementation. Further, SGX provides a paging mechanism significantly increasing the available trusted memory, therefore increasing the possible size of the KVS. While SGX-paging incurs a high overhead, often accessed keys will eventually resided in EPC.

Result. Our evaluation in Section 6.4 confirms that our AVOCADO single-node KVS is scalable and more performant; the speedup of AVOCADO single-node KVS compared to ShieldStore increases from 1.6× in a single threaded benchmark to 5× when utilizing all 8 available CPU threads.

5 Implementation

5.1 System components

AVOCADO network stack. The AVOCADO network stack is based on eRPC [46] and DPDK [4]. In particular, we leverage SCONE to build both eRPC and DPDK. We also assure that the device DMA mappings resides in the host memory. The changes to implement the mappings amount to 154 new LoC and 81 removed LoC.

To run eRPC inside the enclave, we accordingly modify the hugepages allocation mechanism (a) to ensure that all network buffers reside in the host memory, (b) to fix a bug regarding the hugepages’ detection, and (c) to alter how the address of the memory region is calculated. We also replace eRPC’s allocation algorithm with our own allocator. We notice that the eRPC’s native allocation algorithm, which allocates double the space of the previous allocation, quickly reserves all available memory in AVOCADO. Our memory

allocator is less aggressive and allows us to use our servers' limited huge page memory more efficiently. In total, 80 LoC are added to eRPC, while 28 LoC are removed.

eRPC provides us with its own implementation of the UDP protocol. To secure the network communication, on top of the layer protocol, we use a modified OpenSSL [6] version. These changes allow us to randomly access the encrypted data. We added 55 LoC to OpenSSL. Further, we added another 287 LoC for a shared en-/decryption layer for the AVOCADO single-node KVS store and networking. Lastly, we further extend the shared layer to well-fit with the message format. This adds another 205 LoC.

AVOCADO replication layer. We implement AVOCADO replication layer in C++ on top of the AVOCADO network stack (2,743 LoC). We implement the protocol from scratch using the eRPC networking library across AVOCADO's different layers, i.e., replication and networking layer.

Configuration and attestation service. We implement AVOCADO CAS in Rust [59] for better memory safety (22,730 LoC). To run the CAS inside the Intel SGX, we use SCONE since it transparently supports Rust applications. We make use of an encrypted embedded SQLite [9] to maintain configurations and secrets of AVOCADO inside AVOCADO CAS. To setup the configuration and bootstrap process, we provide configurations scripts, in Bash and Python 3, in total these bootstrap scripts require 709 LoC.

AVOCADO single-node KVS. We implement the AVOCADO single-node KVS based on a skip list based partitioned data structure. Particularly, AVOCADO single-node KVS extends Folly's ConcurrentSkipList [32]. We ported the Folly library to SCONE, which resulted to 167 new LoC and 40,394 removed LoC. In addition, the implementation requires another 190 LoC for the integration of the Boost library [19] to SCONE. Further, we implement an efficient host memory allocator (388 LoC) for our skip list. We share en-/decryption layer based on OpenSSL [6] with the network stack.

5.2 Optimizations

[O1] Remove duplicated en-/decryptions. In AVOCADO, we use a shared encryption key between all replicas for the network operations. This allows us to replace some encryption calls with memory copies, as we can send the same packets to all replicas without costly re-encrypting the messages. However, this optimization is an optional trade-off between security and performance since one compromised enclave would compromise the entire system.

[O2] Remove locks. Separating the metadata from the key allows us to make atomic updates to the skip list, avoiding expensive locks. However, it also allows us to retire values earlier to the host memory; thereby reducing the EPC pressure since the metadata can already be written without being visible to other calls. Further, our host memory allocator supports lock-free operations on our skip list by providing similar atomic allocation and de-allocation primitives.

[O3] Limited number of message buffers. We design a rate limiter to allow all current running requests to finish without having to wait for the available resources. While we mostly implement it to prevent eRPC from running out of hugepages memory, we also find that it also reduces the stalls between accepting and completing a request.

6 Evaluation

Our evaluation answers the following questions.

- How does the AVOCADO network stack perform compared to the alternative networking approaches? (§ 6.1)
- How does the AVOCADO replication layer compare with alternative protocols (Raft [64] & BFT [87])? (§ 6.2)
- What are the performance overheads of AVOCADO CAS and how it compares with Intel's IAS [3]? (§ 6.3)
- How does the AVOCADO single-node KVS perform compared to ShieldStore [48]? (§ 6.4)
- What are the overall performance overheads of AVOCADO KVS? (§ 6.5)
- How does AVOCADO scale with increasing number of nodes? (§ 6.6)

Testbed. We perform all of our experiments on real hardware using a cluster of 5 SGX server machines with CPU: Intel(R) Core(TM) i9-9900K each with 8 cores (16 HT), memory: 64 GiB, caches: 32 KiB (L1 data and code), 256 KiB (L2) and 16 MiB (L3), NIC: Intel Corporation Ethernet Controller XL710 for 40GbE QSFP+ (rev 02). They are connected over a 40GbE QSFP+ network switch.

Benchmarks. For the evaluation, we use the YCSB benchmark [7, 24] with different read/write ratios. Client-server communication over the network is prohibitively expensive from within an enclave (see § 4.1). Therefore, we stress-test the performance of AVOCADO by generating the workload within the enclave. This is the worst-case scenario for our system, since a client-server setting will show negligible latency/throughput overheads, due to client-server communication being the bottleneck. We configured AVOCADO to use a shared network key between the replicas (§ 5.2[O1]). For evaluating the network stack, we use iperf [41].

6.1 Network stack

Baselines and setup. We evaluate the performance of the AVOCADO network stack against three competitive baselines: eRPC-native, sockets-native, and sockets-SCONE. Note that SCONE uses asynchronous syscalls [85] for performance improvements. Further, note that the native (eRPC and sockets) versions do not provide any security.

For the sockets (native and SCONE), we use iperf to measure the throughput. For eRPC-native and AVOCADO network stack, we implement a simple server-client model on top of eRPC to simulate iperf's behavior.

In our experiments, we compare the performance with different number of packet sizes, while keeping the number of threads fixed to 4. Note that eRPC supports only UDP while

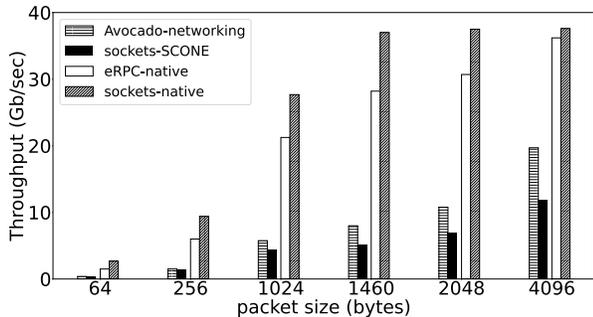


Figure 6: Performance comparison of AVOCADO network stack, eRPC-native, sockets-native, and sockets-SCONE for different packet sizes.

iperf supports both TCP and UDP. In our servers, we found that TCP performs better than UDP, so we report iperf’s TCP measurements since a designer could always benchmark both protocols and choose the most performant one.

Results. Figure 6 shows that eRPC-native is comparable to sockets-native. The reason is that TCP is optimized for high speed bulk transfers while UDP is optimized for low latency in the Linux kernel. This has an impact on buffer sizes and how data is polled and handed over. In addition to this, processing of the entire TCP/IP stack is frequently offloaded to the network controller.

Based on Figure 6 we deduce two core conclusions; (a) SCONE’s overhead is not negligible—SCONE performance degrades $\sim 4\times$ and $\sim 8\times$ compared to AVOCADO network stack and sockets-SCONE, respectively; and (b), due to the number of system calls the sockets’ layer is executing, AVOCADO network stack in the context of the secure enclave performs up to $1.66\times$ faster than socket s-SCONE. As discussed, enclave exits and data copies in and out of the enclave deteriorate sockets’ performance. This is further supported by the fact that the bigger the packet size is, the worse the performance becomes. Therefore, sockets-SCONE is a poor design choice as far as our requirement is concerned, and it justifies our design of the AVOCADO network stack.

6.2 Replication protocol

Baselines and setup. We show our system’s end-to-end performance in comparison with two state-of-the-art protocols: (a) BFT (BFT-Smart [87]) for the Byzantine setting, and (b) Raft (eRPC-Raft [1]) for the non-Byzantine setting. To the best of our knowledge, there is no secure distributed in-memory KVS; BFT-Smart KVS is the closest baseline in terms of security properties for Byzantine environments, but BFT protocols (or BFT-Smart) still do not preserve confidentiality. Additionally, we compare AVOCADO against eRPC-Raft since it is also built on top of eRPC. This comparison aims to demonstrate the efficacy of eRPC. We compare them with a native version of AVOCADO, AVOCADO-native, which runs without TEEs. We compare AVOCADO and BFT along three parameters, as shown in Figure 7; (i) different

	kOp/s	Speedup
AVOCADO	96	5.05 \times
eRPC-Raft	19	

Table 2: Performance comparison between AVOCADO and eRPC-Raft under a 100% W workload and a single client.

read/write ratios, (ii) different value sizes and (iii) different workload threads per machine. We evaluate using the YCSB benchmark [7, 24]. Similarly, we compare AVOCADO against Raft implemented with eRPC. Note that eRPC-Raft is limited to only PUT requests and 1 workload thread in total.

Results. Our evaluation shows that AVOCADO can achieve $4.5\times$ to $65\times$ more operations per second compared to BFT. Our AVOCADO presents similar performance to all four workloads, deducting that it is equivalently performant to both read and write heavy workloads. In addition, we notice that striving for the strictest security guarantees can decrease the performance to half compared to a native, unsecure version of AVOCADO.

Furthermore, we observe that the value size has great impact in the end-to-end performance. For instance, even in a read-heavy workload with value size to be equal to 256 B, the performance of AVOCADO is $6\times$ higher compared to BFT and $1.83\times$ lower than the native version. However, for value size to be equal to 1024 B, AVOCADO is 20% slower than BFT and $9\times$ slower than AVOCADO-native. Similarly, for value size to be equal to 4096 B, AVOCADO is $1.25\times$ faster than BFT and $3.65\times$ slower than AVOCADO-native. We discuss the effects of value size on AVOCADO further in section § 6.5. Lastly, AVOCADO scales up with the number of threads; AVOCADO achieves 38% more operations when the number of threads is increased from 4 to 8 threads. Due to the limitation with the amount of threads inside the enclave, AVOCADO cannot be executed with 16 threads.

Lastly, we compare eRPC-Raft against AVOCADO. AVOCADO under the same settings outperforms eRPC-Raft for $4.8\times$ as shown in Table 2. The reason is that eRPC-Raft does process requests asynchronously while in AVOCADO the time required for the necessary replicas to respond overlaps with processing any outstanding requests.

6.3 Configuration and attestation service

Baseline and setup. To evaluate the advantage of the attestation mechanism using AVOCADO CAS in comparison to the traditional attestation mechanism of Intel using IAS, we conduct an experiment to measure the end-to-end latency of the attestation process using both mechanisms.

Results. The attestation using AVOCADO CAS achieves a speedup of $18.2\times$ compared to the traditional mechanism using IAS (see Table 3). The mechanism using AVOCADO CAS performs the attestation via LAN connections, since AVOCADO CAS is deployed in the same cluster as AVOCADO instances. Meanwhile, the mechanism using IAS performs the attestation via WAN connections since it requires to verify the quotes using IAS that is deployed at Intel. Furthermore, AVOCADO CAS

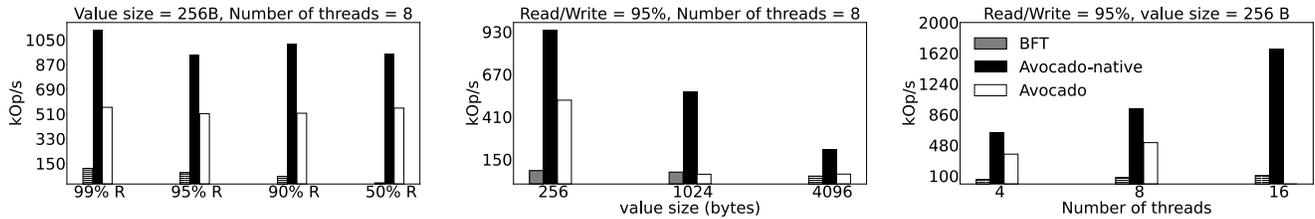


Figure 7: End-to-end performance comparison between AVOCADO, AVOCADO-native and BFT for different types of workloads, value sizes and number of threads.

	Mean / s	SD / s	Speedup
AVOCADO CAS	0.169	0.0195	18.2×
IAS	2.913	0.177	

Table 3: The end-to-end latency comparison between the attestation mechanisms using AVOCADO CAS and IAS.

provides transparently provides configuration management features to operate in an distributed environment.

6.4 Single-node KVS

Baselines and setup. We compare our AVOCADO single-node KVS against ShieldStore [48], a state-of-the-art secure in-memory KVS for a single-node system. For the single-node system evaluation, we use Intel(R) Core(TM) i7-8565U CPU of 8 logical threads and 16 GiB memory. This is due to ShieldStore’s dependencies on specific versions of OS, linux-SGX [82] and tcmlloc [58], we are not able to run it on our servers. We evaluate AVOCADO single-node KVS and ShieldStore across three dimensions using the YCSB workload: threads (Figure 8), value size and read-write ratios (Figure 9).

Results. Figure 8 shows the scaling capabilities of our AVOCADO single-node KVS vs. ShieldStore for two different value sizes. Our AVOCADO single-node KVS, for all number of threads, is 1.6× to 9.5× faster than ShieldStore. Regarding the value size, we observe that ShieldStore’s performance is highly affected when the value size is increased. For example, with 2 threads, ShieldStore presents a performance degradation of 7.3× from 16 B to 1024 B while the same scenario deteriorates AVOCADO single-node KVS’s performance only by 1.23×. We deduct this to the fact that Shieldstore searches require decrypting the concatenated entry of the key and the value in each visited bucket. As a result, bigger value sizes increase the cipher text that needs to be decrypted leading to higher performance costs. In contrast, AVOCADO stores keys inside protected area and search time is not affected by value sizes and decryption cost.

We observe similar behavior as the number of threads increases. ShieldStore is designed to avoid synchronization costs between threads that are matched to different KVS’s areas. However, to achieve this, ShieldStore requires to sort and distribute (through hashing) requests across threads which adds overheads compared to AVOCADO single-node KVS. Specifically, we show that for value size equal to 16 B AVOCADO single-node KVS is 1.6×, 3× and 5× faster than ShieldStore when using

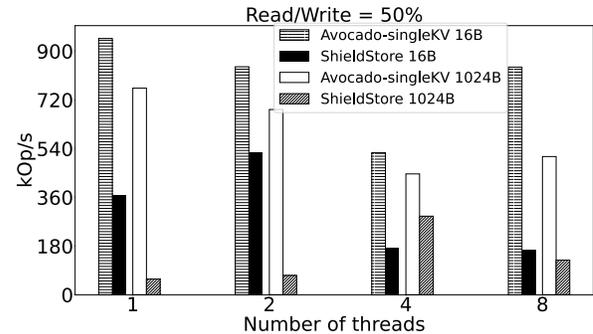


Figure 8: Performance comparison between AVOCADO single-node KVS and ShieldStore under a 50R-50W workload for varying number of threads.

2, 4 and 8 threads, respectively. Additionally, for value size equal to 1024 B, AVOCADO single-node KVS is 9.5×, 1.5× and 4× faster than ShieldStore with 2, 4 and 8 threads, respectively.

However, we find that both AVOCADO single-node KVS and ShieldStore have a performance drop, when the number of threads is increased. This trend is visible until the number of threads exceeds the number of physical cores. We attribute this behavior to two main reasons. Firstly, the CPU we are running this experiment on, is a lower power CPU, with a low base frequency (1.8 GHz) but a relatively high turbo frequency (4.8 GHz). This high turbo frequency cannot be reached with a high number of threads running, giving a performance boost to low thread numbers, compared to higher thread numbers. Secondly, increasing the number of threads results in a higher rate of cache misses, due to other cores having requested different memory, or having to invalidate the cache lines in lower level caches i.e. L1 and L2. This effect is especially pronounced in a write heavy workload, as presented in Figure 8. Increasing the number of threads further, from number of physical cores to logical cores, allows the CPU to schedule a different thread, while it is waiting for a memory access to complete, explaining the increase of performance with 8 threads.

Secondly, we also study how AVOCADO single-node KVS and ShieldStore perform under different value sizes and different read-write ratios. In particular, Figure 9 compares the two Key-Value (KV) stores against three different workloads and varying value sizes, where we fix the number of threads across the experiments to 4. For all three workloads as shown in Figure 9, AVOCADO single-node KVS achieves better performance than ShieldStore; 3.63× to 2.97× faster when value size is equal

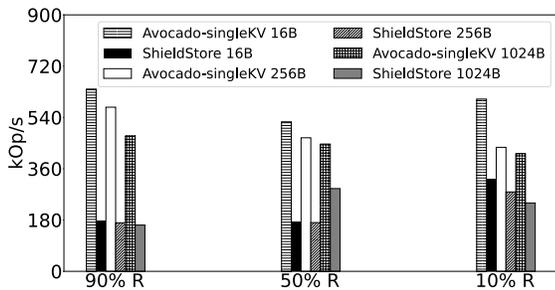


Figure 9: Performance comparison between AVOCADO single-node KVS and Shieldstore under three different workloads for varying value sizes.

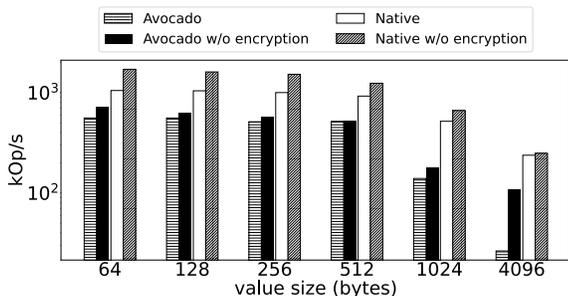


Figure 10: Performance comparison of AVOCADO with and without network and KVS encryption, inside and outside of the enclave, with different value sizes with 95% reads and 8 threads per machine.

to 16 B, $3\times$ to $1.53\times$ faster when value size is equal to 256 B and $1.87\times$ to $1.56\times$ faster when value size is equal to 1024 B.

Lastly, we conclude that AVOCADO single-node KVS is better in read-dominant workload (90% reads) than in write-heavy workload (90% writes), since AVOCADO single-node KVS achieves $\sim 5\%$ to $\sim 30\%$ better performance.

6.5 Distributed KVS

Baselines and setup. We evaluate the overhead AVOCADO incurs from running inside an enclave, against running AVOCADO natively, i.e. without SGX. Furthermore, we also evaluate the encryption overheads for in-memory KVS and network traffic. Thus, we compare AVOCADO with encryption activated and deactivated against the native KVS. Both with encryption for the KVS and network enabled and disabled. We run the YCSB benchmark with 95% reads with 5 machines and 8 threads per machine, with different value sizes.

Results. Figure 10 shows the performance influence of SGX and encryption on AVOCADO. The value size comparison shows that for small values, i.e. values under <1 KiB, AVOCADO reaches around half, between 51.2 and 56.0% of the performance compared to the native KVS. However, with bigger value sizes the difference becomes greater with a slowdown of $3.7\times$ and $9.0\times$, for 1 or 4 KiB respectively. The sudden drop in performance compared to the native case is mainly due to two reasons: firstly, eRPC has to acquire a lock

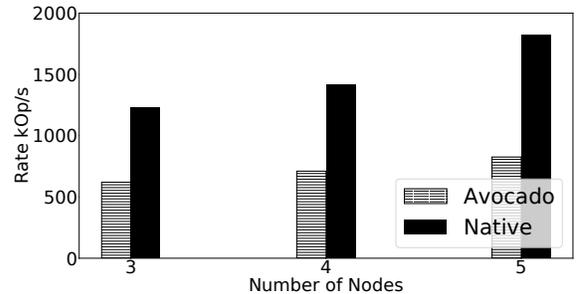


Figure 11: Performance of AVOCADO inside and outside of the enclave running on different number of nodes with a value size of 256 B, 95% reads and 8 threads per machine.

when allocating DMA for bigger packages size than the MTU, which is configured in our case to 1 KiB. While this penalizes native and AVOCADO, it is worse for AVOCADO, since this could result in an enclave exit for yielding. Additionally, with bigger value sizes, it gets more likely that we have to evict a page from the EPC, when inspecting the network traffic. This might be addressed with a memory buffer, which is reused for all data transfer between untrusted host memory and EPC memory. Due to constant accessing of this buffer, it should rarely get paged out the enclave.

The comparison also shows that encrypting the in-memory KVS and network traffic adds up to 62% overhead for small values and 4.6% for large values in the native case. However, we observe a different behavior for AVOCADO. The overhead for small values is more moderate compared to native with around 25%. However, the overhead does not get smaller with bigger values sizes. In contrast, it peaked with a values size of 4 KiB with $4.1\times$, which is due to EPC paging.

In these experiment we also observed a mean latency of $66\ \mu\text{s}$. This latency was reached in a fully stressed system. Due to SGX requiring us to make a syscall for taking a timestamp detailed latency analysis was impractical, as the syscall overhead together with the world switch would have easily dominated the benchmark.

6.6 Scalability

Baselines and setup. We evaluated the scalability of AVOCADO by running it inside and outside (natively) the enclave on a varying number of nodes. We run the YCSB benchmark with 95% reads on 3, 4 and 5 machines and 8 threads per machine, with a fixed value size of 256 B.

Results. Figure 11 shows the scalability numbers for different number of nodes. We are limited in our cluster to 5 nodes. The evaluation shows that replicating the KVS over 5 nodes instead of 3 increases the throughput of the native solution by 49% and 33% for AVOCADO.

6.7 Number of keys

Baselines and setup. We measure the performance of AVOCADO with increasing number of distinct keys. We run the YCSB benchmark with two different distributions, i.e.

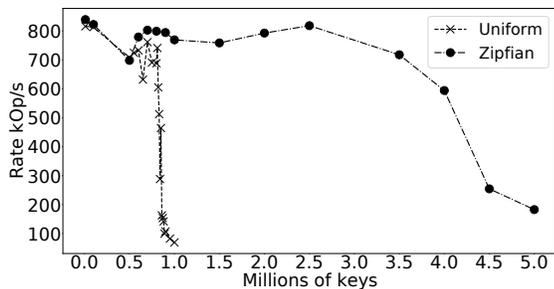


Figure 12: Performance of AVOCADO inside the enclave running with different number of distinct keys, with a uniform and Zipfian ($\alpha = 1.0$) distribution with a value size of 256 B, 95% reads and 8 threads per machine

uniform and Zipfian, on 5 machines with 8 threads and 95% reads and a fixed value size of 256 B.

Results. Figure 12 shows the throughput of AVOCADO with different distribution. Both distributions have a similar performance until 700k keys, where the performance of the uniform distribution starts to suffer greatly, due to SGX paging overheads. The uniform distribution prevents efficient caching from SGX in the EPC, since it does not generate any hot keys. In a uniform distribution, AVOCADO is therefore restricted by the EPC size. However, this could be alleviated with a multi-level lookup, which stores the lower levels in the host memory.

On the other hand, if the data set is not uniformly distributed AVOCADO can take advantage of the caching of EPC and extend the number of supported keys. In our experiments AVOCADO throughput was stable until 3.5M keys in the Zipfian distribution before it starts to suffer from the paging overheads. Similar to the uniform distribution, a multi-level lookup could reduce the paging overheads.

7 Related work

Shielded execution. With the adoption of TEEs in the cloud, shielded execution frameworks are being adopted to provide strong security properties for unmodified/legacy applications running in the untrusted environment [13, 17, 35, 93]. These frameworks promote portability, programmability and good performance. As a result, they have been used to implement a wide-range of secure systems for storage [16, 50], data analytics [80], databases [73], distributed systems [90], FaaS [92], network functions [91], machine learning [75], profilers [15], etc. Our AVOCADO project leverages the advancements in shielded execution frameworks; in particular, we use SCONE [13] to build a distributed storage system.

Networking for shielded execution. Shielded execution frameworks, like SCONE [13] and Eleos [65] provide an asynchronous system call API [85]. However, the asynchronous syscall mechanism incurs high overheads (due to data copies) and latency. ShieldBox [91] alleviates the issue of asynchronous syscalls by using DPDK [4] as polling user mode driver for a secure middlebox. Unfortunately, Shieldbox network stack targets only layer 2 in the OSI model, and

therefore, it is incompatible with the RPC layer required for a distributed KVS. rkt-io provides a library OS in the enclave, and can therefore provide a full network stack [89].

Secure storage systems. Secure storage is an important theme for cloud computing systems. A wide-range of systems have been proposed in the community based on different hardware with varying security guarantees and storage interfaces: KVS [16, 48, 50], filesystems [33, 96, 103], databases [28, 67, 70, 73, 95], etc. In contrast to these existing systems, AVOCADO proposes a scalable distributed in-memory KV store instead of a single-node system.

For distributed storage system design, Depot [57] and Salus [102] propose secure distributed storages, which provide consistency, durability, availability and integrity. A2M [21] is also robust against Byzantine faults. On top of those properties, AVOCADO also offers confidentiality. CloudProof [69] completely distrusts the cloud provider. However, CloudProof requires the client to guarantee these security properties, which requires major changes to the client, which isn't required by AVOCADO. Furthermore, since our work leverages hardware-assisted shielded execution as the root of trust, we do not need a trusted proxy, which limits the scalability of the system.

8 Conclusion

We present an approach to build a secure, high-performance in-memory distributed KVS system for untrusted cloud environments using TEEs. Our design includes four core contributions involving TEEs in a distributed environment: (a) the first direct I/O RPC network stack for TEEs based on eRPC with the complete support for transport and session layers; (b) a secure replication protocol based on hardening of a non-Byzantine protocol, where we transform a Byzantine behavior to a faulty behavior using TEEs; (c) a configuration and attestation service to seamlessly extend the trust from a single-node TEE to the distributed environment; (d) a secure in-memory single-node KVS based on a novel partitioned Skiplist data structure, and show that it is well-suited to overcome the memory limitations and support lock-free scalable concurrent updates in the TEEs.

Importantly, we set out to build a practical system without compromising performance—the literature distinctly shows that BFT protocols are typically not adopted in practice due to their high overheads [22, 71]. In contrast to BFT, our system provides stronger security properties (also preserves confidentiality) and improved performance (4.5× to 65×), while using f fewer replicas.

Software artifact. Our project is publicly available: <https://github.com/mbailleu/avocado>.

Acknowledgements. We thank our shepherd, Yu Hua, and the anonymous reviewers for their helpful comments. This work was supported in parts by a UK RISE Grant from NCSC/GCHQ, a Huawei Research Grant, and a Microsoft Research PhD Fellowship.

References

- [1] eRPC-Raft. <https://github.com/erpc-io/eRPC/tree/master/apps/smr>. Last accessed: Jan, 2021.
- [2] How long does it take to make a context switch? <https://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html>. Last accessed: Jan, 2021.
- [3] Intel Corporation. Attestation Service for Intel Software GuardExtensions (Intel SGX): API Documentation. <https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf>. Last accessed: Jan, 2021.
- [4] Intel DPDK. <http://dpdk.org/>. Last accessed: Jan, 2021.
- [5] Intel Software Guard Extensions (Intel SGX). <https://software.intel.com/en-us/sgx>. Last accessed: Jan, 2021.
- [6] OpenSSL library. <https://openssl.org>. Last accessed: Jan, 2021.
- [7] YCSB. <https://github.com/brianfrankcooper/YCSB>. Last accessed: Jan, 2021.
- [8] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(6):159–174, Oct. 2007.
- [9] G. Allen and M. Owens. *The Definitive Guide to SQLite*. Apress, 2010.
- [10] AMD. AMD Secure Encrypted Virtualization (SEV). <https://developer.amd.com/sev/>. Last accessed: Jan, 2021.
- [11] I. Anati, S. Gueron, P. S. Johnson, and R. V. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.
- [12] ARM. Building a secure system using trustzone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. Last accessed: Jan, 2021.
- [13] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O’Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
- [14] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message-passing Systems. *J. ACM*, 42(1), Jan. 1995.
- [15] M. Bailieu, D. Dragoti, P. Bhatotia, and C. Fetzer. Teeperf: A profiler for trusted execution environments. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019.
- [16] M. Bailieu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. SPEICHER: Securing lsm-based key-value stores using shielded execution. In *17th USENIX Conference on File and Storage Technologies (FAST)*, 2019.
- [17] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.
- [18] J. Behl, T. Distler, and R. Kapitza. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017.
- [19] boost: C++ libraries. <https://www.boost.org/>. Last accessed: Aug, 2020.
- [20] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 2002.
- [21] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.
- [22] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. Bft: The time is now. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2008.
- [23] A. Cloud. Alibaba Cloud’s Next-Generation Security Makes Gartner’s Report. https://www.alibabacloud.com/blog/alibaba-clouds-next-generation-security-makes-gartners-report_595367. Last accessed: Jan, 2021.
- [24] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud computing (SoCC)*, 2010.
- [25] M. Correia, N. Neves, and P. Verissimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, 2004.

- [26] M. Correia, N. F. Neves, L. C. Lung, and P. Verissimo. Worm-IT - A Wormhole-Based Intrusion-Tolerant Group Communication System. 2007.
- [27] V. Costan and S. Devadas. Intel SGX Explained, 2016.
- [28] N. Crooks, M. Burke, E. Cecchetti, S. Harel, R. Agarwal, and L. Alvisi. Obladi: Oblivious Serializable Transactions in the Cloud. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2018.
- [29] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchun, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon’s highly available key-value store. *ACM SIGOPS Operating Systems Review (SIGOPS)*, 2007.
- [30] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.
- [31] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004.
- [32] Folly: Facebook Open-source Library. <https://github.com/facebook/folly>.
- [33] D. Garg and F. Pfenning. A proof-carrying file system. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.
- [34] Introducing Google Cloud Confidential Computing with Confidential VMs. <https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms>.
- [35] Asylo: An open and flexible framework for enclave applications. <https://asylo.dev/>.
- [36] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014.
- [37] M. Hähnel, W. Cui, and M. Peinado. High-resolution side channels for untrusted operating systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.
- [38] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.
- [39] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transaction of Programming Language and Systems (TOPLAS)*, 1990.
- [40] J. Huang, X. Ouyang, J. Jose, M. Wasi-ur-Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. K. Panda. High-Performance Design of HBase with RDMA over InfiniBand. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.
- [41] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. <https://iperf.fr/>. Last accessed: Aug, 2020.
- [42] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.
- [43] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2014.
- [44] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur-Rahman, H. Wang, S. Narravula, and D. K. Panda. Scalable Memcached Design for InfiniBand Clusters Using Hybrid Transports. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid)*, 2012.
- [45] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur-Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *International Conference on Parallel Processing (ICPP)*, 2011.
- [46] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.
- [47] A. Katsarakis, V. Gavrielatos, M. S. Katebzadeh, A. Joshi, A. Dragojevic, B. Grot, and V. Nagarajan. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.
- [48] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh. ShieldStore: Shielded In-Memory Key-Value Storage with SGX. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys)*, 2019.

- [49] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P)*, 2019.
- [50] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer. Pesos: Policy enhanced secure object store. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, 2018.
- [51] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the 12th ACM European Conference on Computer Systems (EuroSys)*, 2017.
- [52] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 1982.
- [53] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. Keystone: an open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*, 2020.
- [54] D. Levin, J. J. Douceur, J. Lorch, and T. Moscibroda. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.
- [55] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
- [56] N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing (FtCS)*, 1997.
- [57] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud Storage with Minimal Trust. In *ACM Transactions on Computer Systems*, 2011.
- [58] TCMalloc. <https://github.com/google/tcmalloc>. Last accessed: Aug, 2020.
- [59] N. D. Matsakis and F. S. Klock, II. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT)*, 2014.
- [60] Microsoft Azure. Azure confidential computing. <https://azure.microsoft.com/en-us/solutions/confidential-compute/>. Last accessed: Jan, 2021.
- [61] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (ATC)*, 2013.
- [62] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.
- [63] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.
- [64] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2014.
- [65] M. Orenbach, M. Minkin, P. Lifshits, and M. Silberstein. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the 12th ACM European ACM Conference in Computer Systems (EuroSys)*, 2017.
- [66] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. 2015.
- [67] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan. Big data analytics over encrypted datasets with seabed. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
- [68] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping Trust in Commodity Computers. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P)*, 2010.
- [69] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage slas with cloudproof. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC)*, 2011.
- [70] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

- [71] D. Porto, J. a. Leitão, C. Li, A. Clement, A. Kate, F. Junqueira, and R. Rodrigues. Visigoth fault tolerance. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)*, 2015.
- [72] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch. Sgx-kl: Securing the host os interface for trusted execution, 2019.
- [73] C. Priebe, K. Vaswani, and M. Costa. EnclaveDB: A Secure Database using SGX (S&P). In *IEEE Symposium on Security and Privacy*, 2018.
- [74] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communication of ACM (CACM)*, 1990.
- [75] D. L. Quoc, F. Gregor, S. Arnautov, R. Kunkel, P. Bhatotia, and C. Fetzer. Securetf: A secure tensorflow framework. In *Proceedings of the 21st International Middleware Conference (Middleware)*, 2020.
- [76] RISC-V. Keystone Open-source Secure Hardware Enclave. <https://keystone-enclave.org/>.
- [77] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards Trusted Cloud Computing. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.
- [78] N. Santos, R. Rodrigues, and B. Ford. Enhancing the os against security threats in system administration. In *Proceedings of the 13th International Middleware Conference (Middleware)*, 2012.
- [79] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services . In *21st USENIX Security Symposium (USENIX Security)*, 2012.
- [80] F. Schuster, M. Costa, C. Gkantsidis, M. Peinado, G. Mainar-ruiz, and M. Russinovich. VC3 : Trust-worthy Data Analytics in the Cloud using SGX. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, 2015.
- [81] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019.
- [82] Intel Software Guard Extensions SDK for Linux. <https://01.org/intel-softwareguard-extensions>.
- [83] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCCS)*, 2016.
- [84] A. K. Simpson, A. Szekeres, J. Nelson, and I. Zhang. Securing RDMA for High-Performance Datacenter Storage Systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2020.
- [85] L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.
- [86] L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.
- [87] J. Sousa and A. Bessani. From Byzantine Consensus to BFT State Machine Replication: A Latency-Optimal Transformation. In *2012 Ninth European Dependable Computing Conference (EDCC)*, 2012.
- [88] Y. Taleb, R. Stutsman, G. Antoniu, and T. Cortes. Tailwind: Fast and Atomic RDMA-Based Replication. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2018.
- [89] J. Thalheim, H. Unnibhavi, C. Priebe, P. Bhatotia, and P. Pietzuch. Rkt-io: A direct i/o stack for shielded execution. In *Proceedings of the Sixteenth European Conference on Computer Systems (ACM EuroSys 21)*, 2021.
- [90] B. Trach, R. Faqeh, O. Oleksenko, W. Ozga, P. Bhatotia, and C. Fetzer. T-lease: A trusted lease primitive for distributed systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*, 2020.
- [91] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. ShieldBox: Secure Middleboxes using Shielded Execution. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2018.
- [92] B. Trach, O. Oleksenko, F. Gregor, P. Bhatotia, and C. Fetzer. Clemmys: Towards secure remote execution in faas. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR)*, 2019.
- [93] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2017.
- [94] S.-Y. Tsai and Y. Zhang. A Double-Edged Sword: Security Threats and Opportunities in One-Sided Network Communication. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019.
- [95] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the 39th international conference on Very Large Data Bases (VLDB)*, 2013.

- [96] A. Vahldiek-Oberwagner, E. Elnikety, A. Mehta, D. Garg, P. Druschel, R. Rodrigues, J. Gehrke, and A. Post. Guardat: Enforcing data policies at the storage layer. In *Proceedings of the 10th ACM European Conference on Computer Systems (EuroSys)*, 2015.
- [97] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.
- [98] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6 (OSDI)*, 2004.
- [99] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom. SGAXe: How SGX fails in practice. <https://sgaxeattack.com/>, 2020.
- [100] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom. CacheOut: Leaking Data on Intel CPUs via Cache Evictions, 2020.
- [101] V. Vasudevan, D. Andersen, and M. Kaminsky. The Case for VOS: The Vector Operating System. In *13th Workshop on Hot Topics in Operating Systems (HotOS)*, 2011.
- [102] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubananadam, L. Alvisi, and M. Dahlin. Robustness in the salus scalable block store. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.
- [103] C. Weinhold and H. Härtig. jVPFS: Adding Robustness to a Secure Stacked File System with Untrusted Local Storage Components. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2011.
- [104] O. Weisse, V. Bertacco, and T. Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. *SIGARCH Comput. Archit. News*, 2017.
- [105] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, 2015.
- [106] S. Yang. SLIK : Scalable Low-Latency Indexes for a Key-Value Store. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2014.

Accelerating Encrypted Deduplication via SGX

Yanjing Ren[†], Jingwei Li^{†*}, Zuoru Yang[‡], Patrick P. C. Lee[‡], and Xiaosong Zhang[†]

[†]University of Electronic Science and Technology of China [‡]The Chinese University of Hong Kong

Abstract

Encrypted deduplication preserves the deduplication effectiveness on encrypted data and is attractive for outsourced storage. However, existing encrypted deduplication approaches build on expensive cryptographic primitives that incur substantial performance slowdown. We present SGXDedup, which leverages Intel SGX to speed up encrypted deduplication based on server-aided message-locked encryption (MLE), while preserving security via SGX. SGXDedup implements a suite of secure interfaces to execute MLE key generation and proof-of-ownership operations in SGX enclaves. It also proposes various designs to support secure and efficient enclave operations. Evaluation on synthetic and real-world workloads shows that SGXDedup achieves significant speedups and maintains high bandwidth and storage savings.

1 Introduction

Outsourcing storage management to the cloud is a common practice for clients (enterprises or individuals) to save the overhead of self-managing massive data, and *security* and *storage efficiency* are two major goals for practical outsourced storage. To satisfy both goals, we explore *encrypted deduplication*, a paradigm that combines encryption and deduplication by always encrypting duplicate plaintext chunks (from the same or different clients) into duplicate ciphertext chunks with a key derived from the chunk content itself; for example, the key can be the cryptographic hash of the corresponding chunk [25]. Thus, any duplicate ciphertext chunk can be eliminated via deduplication for storage efficiency, while all outsourced chunks are encrypted against unauthorized access. Encrypted deduplication is particularly suitable for backup applications, which carry high content redundancy [59] and are attractive use cases for outsourced storage [35, 38, 58].

Existing encrypted deduplication approaches often incur high performance overhead to achieve security guarantees. We use DupLESS [14], a state-of-the-art encrypted deduplication system, as a representative example to explain the performance concerns (see §2.1 for elaboration). First, to prevent adversaries from inferring the content-derived keys, DupLESS employs *server-aided key management*, in which it deploys a dedicated key server to manage key generation requests from clients. However, server-aided key management requires expensive cryptographic operations to prevent the key server from knowing the plaintext chunks and the keys during key generation. Second, to prevent adversaries

from obtaining unauthorized access to ciphertext chunks by inferring the deduplication pattern (a.k.a. side-channel attacks [32, 33]), DupLESS can take one of the following approaches: it either (i) performs *target-based deduplication* (i.e., uploading all ciphertext chunks and letting the cloud remove any duplicate ciphertext chunks) so that the deduplication pattern is protected from any (malicious) client, or (ii) performs *source-based deduplication* (i.e., removing all duplicate ciphertext chunks on the client side without being uploaded to the cloud) and additionally proves to the cloud that it is indeed the owner of the ciphertext chunks (i.e., has access to the full contents of the corresponding plaintext chunks) and is authorized to perform deduplication on the ciphertext chunks. The former incurs extra communication bandwidth for uploading duplicate ciphertext chunks, while the latter incurs expensive cryptographic operations for proving that the client is the owner of the ciphertext chunks. Although various protocol designs have been proposed to address the performance issues of encrypted deduplication, they often weaken security [23, 41, 60], add bandwidth overhead [33, 42], or degrade storage efficiency [41, 51, 62] (see §6 for details).

The advances of hardware-assisted trusted execution [1–3, 57] provide new opportunities to improve the performance of encrypted deduplication. In particular, we focus on Intel Software Guarded Extensions (SGX), which provides a *trusted execution environment (TEE)*, called an *enclave*, for processing code and data with confidentiality and integrity guarantees [13]. Given that SGX achieves reasonably high performance with proper configurations [34], we are motivated to offload the expensive cryptographic operations of encrypted deduplication by directly running sensitive operations in enclaves, so as to improve the performance of encrypted deduplication, while maintaining its security, bandwidth efficiency, and storage efficiency.

We propose SGXDedup, a high-performance SGX-based encrypted deduplication system. SGXDedup builds on server-aided key management as in DupLESS [14], but executes efficient cryptographic operations inside enclaves. Realizing the design of SGXDedup has non-trivial challenges. First, it is critical to securely bootstrap enclaves for hosting trusted code and data, yet attesting the authenticity of enclaves incurs significant delays. Second, each client needs to communicate via a secure channel with the enclave inside the key server, yet the management overhead of the secure channels increases with the number of clients. Finally, clients may renew or revoke cloud service subscriptions, so allowing dynamic client authentication is critical. To this end, we implement three

*Corresponding author: Jingwei Li (jwli@uestc.edu.cn)

major building blocks for SGXDedup:

- *Secure and efficient enclave management*: It protects against the compromise of the key server and allows a client to quickly bootstrap an enclave after a restart.
- *Renewable blinded key management*: It generates a blinded key for protecting the communication between the enclave inside the key server and each of the clients based on key regression [30], such that the blinded key is renewable for dynamic client authentication.
- *SGX-based speculative encryption*: It offloads the online encryption/decryption overhead of secure channel management via speculative encryption [27].

We evaluate our SGXDedup prototype using synthetic and real-world [5, 45] workloads. It achieves significant speedups by offloading cryptographic operations to enclaves (e.g., a $131.9\times$ key generation speedup over the original key generation scheme in DupLESS [14]). It also achieves $8.1\times$ and $9.6\times$ speedups over DupLESS [14] for the uploads of non-duplicate and duplicate data, respectively, and has up to 99.2% of bandwidth/storage savings in real-world workloads.

We release the source code of our SGXDedup prototype at: <http://adslab.cse.cuhk.edu.hk/software/sgxdedup>.

2 Background and Problem

We present background details and formal definitions for encrypted deduplication (§2.1) and Intel SGX (§2.2). We also present the threat model addressed in this paper (§2.3).

2.1 Encrypted Deduplication

Basics. We consider *chunk-based deduplication* [45, 59, 63], which is widely deployed in modern storage systems to eliminate content redundancy. It works by partitioning an input file into non-overlapping *chunks*. For each chunk, it computes the cryptographic hash of the chunk content (called the *fingerprint*). It tracks the fingerprints of all currently stored chunks in a *fingerprint index*. It only stores a physical copy of the chunk if the fingerprint is new to the fingerprint index, or treats the chunk as a duplicate if the fingerprint has been tracked, assuming that fingerprint collisions are highly unlikely in practice [17].

Encrypted deduplication extends chunk-based deduplication with encryption to provide both data confidentiality and storage efficiency for outsourced cloud storage. A *client* encrypts each *plaintext chunk* of the input file with some symmetric secret key into a *ciphertext chunk* and stores all ciphertext chunks in the *cloud* (or any remote storage site), which manages deduplicated storage for ciphertext chunks. To support file reconstruction, the client creates a *file recipe* that lists the fingerprints, sizes, and keys of the ciphertext chunks. It encrypts the file recipe with its own master secret key and stores the encrypted file recipe in the cloud.

Message-locked encryption (MLE) [15] formalizes a cryptographic primitive for encrypted deduplication. It specifies

how the symmetric secret key (called the *MLE key*) is derived from the *content* of a plaintext chunk (e.g., its popular instantiation *convergent encryption (CE)* [25] uses the cryptographic hash of a plaintext chunk as the MLE key). Thus, it encrypts duplicate plaintext chunks into duplicate ciphertext chunks, so that deduplication remains viable on ciphertext chunks.

Server-aided MLE. CE is vulnerable to *offline brute-force attacks*, as its MLE key (i.e., the hash of a plaintext chunk) can be publicly derived. Specifically, an adversary infers the input plaintext chunk from a target ciphertext chunk (without knowing the MLE key) by enumerating the MLE keys of all possible plaintext chunks to check if any plaintext chunk is encrypted to the target ciphertext chunk.

Server-aided MLE [14] is a state-of-the-art cryptographic primitive that strengthens the security of encrypted deduplication against offline brute-force attacks. It deploys a dedicated *key server* for MLE key generation. To encrypt a plaintext chunk, a client first sends the fingerprint of the plaintext chunk to the key server, which returns the MLE key via both the fingerprint and a *global secret* maintained by the key server. If the global secret is secure, an adversary cannot feasibly launch offline brute-force attacks; otherwise, if the global secret is compromised, the security reduces to that of the original MLE. Server-aided MLE further builds on two security mechanisms. First, it uses the *oblivious pseudorandom function (OPRF)* [47] to allow a client send “blinded” fingerprints of the plaintext chunks, such that the key server can still return the same MLE keys for identical fingerprints without learning the original fingerprints. Second, it rate-limits the key generation requests from the clients to protect against *online brute-force attacks*, in which malicious clients issue many key generation requests to the key server, in order to find a target MLE key.

Proof-of-ownership. For bandwidth savings, encrypted deduplication can apply source-based deduplication, instead of target-based deduplication, to remove duplicate ciphertext chunks on the client side without being uploaded to the cloud (§1). The client sends the fingerprints of ciphertext chunks to the cloud, which checks if the fingerprints are tracked by the fingerprint index (i.e., the corresponding ciphertext chunks have been stored). The client then uploads only the non-duplicate ciphertext chunks to the cloud. However, source-based deduplication is vulnerable to *side-channel attacks* [32, 33] when some clients are compromised. One side-channel attack is that a compromised client can query the existence of any target ciphertext chunk (e.g., if the ciphertext chunk corresponds to some possible password [33]) by sending the fingerprint of the ciphertext chunk to the cloud, so as to identify the sensitive information from other clients. Another side-channel attack is that a compromised client can obtain unauthorized access to the stored chunks of other clients. Specifically, it can use the fingerprint of any target ciphertext chunk to convince the cloud that it is the owner of the corresponding ciphertext chunk with full access rights [32].

Proof-of-ownership (PoW) [32] is a cryptographic approach that augments source-based deduplication with protection against side-channel attacks, while maintaining the bandwidth savings of source-based deduplication. Its idea is to let the cloud verify that a client is indeed the owner of a ciphertext chunk and is authorized with the full access to the ciphertext chunk. This ensures that a compromised client cannot query for the existence of other clients' chunks. Specifically, in PoW-based source-based deduplication, a client attaches each fingerprint being sent to the cloud with a *PoW proof*, through which the cloud can verify if the client is the real owner of the corresponding ciphertext chunk. The cloud only responds upon the successful proof verification, thereby preventing any compromised client from identifying the ciphertext chunks owned by other clients.

Limitations. Recall from §1 that existing encrypted deduplication implementations require expensive cryptographic protection. Server-aided MLE necessitates the OPRF protocol [47] to protect the fingerprint information against the key server, yet the OPRF protocol involves expensive public-key cryptographic operations. For example, our evaluation (§5.1) shows that the OPRF-based MLE key generation achieves only up to 25 MB/s (Exp#1) and limits the overall encrypted deduplication performance to 20 MB/s (Exp#4). Also, the existing PoW implementation is based on the Merkle-tree protocol [32], which achieves only 37 MB/s (Exp#3) due to frequent hash computations for chunk-level PoW. In a 1 GbE LAN environment, the computational overhead of PoW even *negates* the performance gain of eliminating the uploads of duplicate data in source-based deduplication. Although we can mitigate PoW computations by applying PoW on a per-file basis (i.e., a client proves its ownership of a file), the cloud cannot verify if a chunk belongs to a file under chunk-based deduplication. Existing solutions that improve the performance of server-aided MLE or PoW often sacrifice security [23, 41, 60], bandwidth efficiency [33, 42], or storage efficiency [41, 51, 62] (§6).

2.2 Intel SGX

We explore hardware-assisted trusted execution to mitigate the performance overhead of encrypted deduplication, while preserving security, bandwidth efficiency, and storage efficiency. In this work, we focus on Intel SGX [3], a suite of security-related instructions built into modern Intel CPUs. SGX shields the execution of code and data in a hardware-protected environment called an *enclave*. In the following, we highlight three security features of an enclave related to our work: isolation, attestation, and sealing.

Isolation. An enclave resides in a hardware-guarded memory region called the *enclave page cache (EPC)* for hosting any protected code and data. An EPC comprises 4KB pages, and any in-enclave application can take up to 96 MB [34]. If an enclave has a larger size than the EPC, it encrypts unused pages and evicts them to the unprotected memory. In this

work, we deploy enclaves in each client and the key server to protect sensitive operations (§3.1). We also limit the size of in-enclave contents to mitigate the paging overhead (§3.4).

An enclave provides an interface, namely *enclave call (ECall)*, such that an outside application can issue ECalls to securely access in-enclave contents. Note that ECalls incur context switching overhead for accessing the enclave memory [34]. We reduce the number of ECalls by batching contents for processing (§4).

Attestation. SGX supports *remote attestation* to authenticate a target enclave via a remote entity (e.g., the cloud). In the remote attestation process (see [3] for elaboration), the remote entity needs to contact the attestation service operated by Intel to check the integrity of the enclave information provided by the target enclave. Then the remote entity verifies the target enclave by comparing its enclave information with the trusted configuration expected in the target enclave. We use remote attestation to ensure that the correct code and data are loaded into each enclave in the first bootstrap.

Sealing. SGX protects enclave contents when they are stored outside an enclave via sealing. It uses a secret *sealing key* to encrypt the data before being evicted. The sealing key can be derived from either the *measurement hash* (i.e., a SHA256 hash on the enclave contents) or the author identity of the enclave, so that only the corresponding enclave can access the sealing key and decrypt the sealed data. Since remote attestation incurs significant delays (Exp#7), we use sealing to eliminate remote attestation after the first bootstrap of an enclave (§3.2).

Remarks. We do not consider memory encryption-based TEEs (e.g., AMD SEV [1] and MK-TME [2]), since they need a large trusted computing base and expose a broad attack surface [46]. Also, AMD SEV [1] does not protect memory integrity, and is vulnerable to the attack that a privileged adversary can manipulate encrypted memory pages [46].

2.3 Threat Model

Adversarial capabilities. We start with the server-aided MLE architecture [14] with multiple clients, the key server, and the cloud. Our major security goal is to achieve data confidentiality for outsourced cloud storage [14] against a *semi-honest* adversary that infers the original plaintext chunks via the following malicious actions:

- The adversary can compromise the key server and learn the MLE key generation requests issued by each client. It can also access the global secret to infer the original chunks in outsourced storage via offline brute-force attacks [14].
- The adversary can compromise one or multiple clients and send arbitrary MLE key generation requests to query the MLE keys of some target chunks [14]. It can also launch side-channel attacks against some target chunks [33] (§2.1), so as to infer the original plaintext chunks owned by other non-compromised clients.

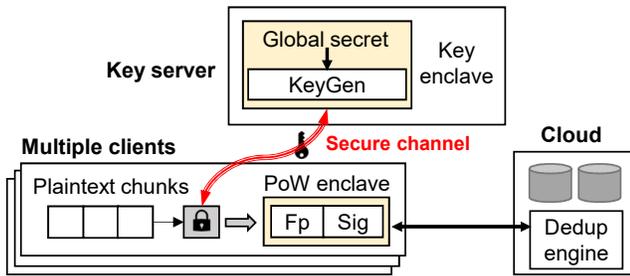


Figure 1: Overview of SGXDedup architecture: a key enclave and a PoW enclave are deployed in the key server and each client, respectively.

Assumptions. We make the following threat assumptions. (i) All communications among the clients, the key server, and the cloud are protected against tampering (e.g., via SSL/TLS). (ii) SGX is trusted and reliable; denial-of-service or side-channel attacks against SGX [18, 48] are beyond our scope. (iii) We can achieve both *integrity* via remote auditing [11, 36] and *fault tolerance* via a multi-cloud approach [42]. (iv) We do not consider traffic analysis [64], frequency analysis [41] and chunk size leakage [54], while the related defenses [41, 54, 64] are compatible to our design.

3 SGXDedup Design

SGXDedup is a high-performance SGX-based encrypted deduplication system designed for outsourced storage, particularly for backup workloads with high content redundancy. It aims for the following design goals: (i) *confidentiality*, which protects the outsourced chunks and keys against unauthorized access even when the key server or any client are compromised, as in server-aided MLE in DupLESS [14] and source-based deduplication with PoW [32]; (ii) *bandwidth/storage efficiency*, which removes all duplicate chunks across multiple clients before uploads, as in source-based deduplication. (iii) *computational efficiency*, which mitigates the computational overhead of the cryptographic operations and achieves significantly higher performance than existing software-based encrypted deduplication designs.

3.1 Overview

Architecture. Figure 1 presents the architecture of SGXDedup, which introduces two enclaves: the *key enclave* and the *PoW enclave*. SGXDedup deploys a key enclave in the key server to manage and protect the global secret of server-aided MLE against a compromised key server. To perform MLE key generation, both the key enclave and a client first establish a secure channel based on a shared *blinded key* (see §3.3 for details how a blinded key is formed). The client then submits the fingerprint of a plaintext chunk via the secure channel. The key enclave computes the MLE key as the cryptographic hash of both the global secret and the fingerprint. It returns the MLE key via the secure channel.

The key enclave benefits both performance and security.

It avoids the expensive OPRF protocol [14] of server-aided MLE during MLE key generation. Also, it protects the fingerprints and MLE keys via a secure channel based on a shared blinded key, such that the key server cannot learn any information from the MLE key generation process. Furthermore, it protects the global secret in the enclave memory, and preserves security even if the key server is compromised (note that the security of the original server-aided MLE degrades when the key server is compromised; see §2.1).

SGXDedup deploys a PoW enclave in each client to prove the authenticity of the ciphertext chunks in source-based deduplication. The PoW enclave first sets up a shared *PoW key* with the cloud (we currently implement the key agreement using Diffie-Hellman key exchange (DHKE); see §4). After MLE key generation, the client encrypts each plaintext chunk into a ciphertext chunk. The PoW enclave takes the ciphertext chunk as input, computes the corresponding fingerprint, and creates a signature of the fingerprint using the shared PoW key with the cloud. The client then uploads both the fingerprint and the signature to the cloud. The cloud verifies the authenticity of the fingerprint based on the corresponding signature and the PoW key. Only when the fingerprint is authenticated, the cloud proceeds to check if the fingerprint corresponds to any already stored duplicate ciphertext chunk. Note that we verify the client’s ownership of ciphertext chunks rather than that of plaintexts (e.g., [32]), so as to protect the original information from the cloud. Ensuring the ownership of ciphertext chunks is enough for security, since MLE applies one-to-one mappings and the ownership of a ciphertext chunk is consistent with that of the corresponding plaintext chunk.

The PoW enclave again benefits both performance and security. It avoids the computational overhead for the cryptographic PoW constructions. It also protects the deduplication patterns against malicious clients, as the patterns are only returned given the authenticated fingerprints.

Note that prior studies [24, 31, 37] perform key generation and encryption inside an enclave, while we opt to perform encryption in unprotected memory. The main reason is that both the original plaintext chunks and the encryption process are co-located within a client. Moving the encryption process to the enclave does not improve the security, as compromising the client can also access its plaintext chunks, yet it adds significant computational overhead to the enclave.

Questions. Realizing SGX-based encrypted deduplication efficiently is non-trivial, since we need to mitigate the potential performance overhead of SGX that would otherwise negate the overall performance benefits. Here, we pose three questions, which we address based on a suite of ECalls for the key enclave and the PoW enclave (Table 1).

- How should the enclaves be securely and efficiently bootstrapped? (§3.2)
- How should the key enclave and each client establish a secure channel? (§3.3)

ECall Name	Description
Key enclave	
<i>Secret generation</i>	Generate a global secret (§3.2)
<i>Rekeying</i>	Renew a blinded key (§3.3)
<i>Nonce checking</i>	Check the uniqueness of a nonce (§3.4)
<i>Key generation</i>	Return the MLE keys (§3.4)
<i>Mask generation</i>	Pre-compute masks (§3.4)
PoW enclave	
<i>Key unsealing</i>	Unseal a PoW key (§3.2)
<i>Key sealing</i>	Seal a PoW key into disk (§3.2)
<i>Proof generation</i>	Sign the ciphertext chunk fingerprints (§4)

Table 1: Major ECalls in SGXDedup.

- How should the key enclave reduce its computational overhead of managing the secure channels of clients? (§3.4)

3.2 Enclave Management

SGXDedup establishes trust in all enclaves via the cloud when it is first initialized. Before SGXDedup is deployed, we first compile the enclave code into shared objects [3], append each shared object with a signature (for integrity verification), and distribute the shared objects to the key server and each of the clients. The cloud also hosts the shared objects for subsequent verification. The key server creates the key enclave, while each client creates its own PoW enclave by loading the corresponding shared object. The cloud authenticates each enclave via remote attestation (§2.2) to ensure that the correct code is loaded. Here, we address two specific management issues: (i) how the global secret (§3.1) is securely bootstrapped into the key enclave; and (ii) how each client efficiently bootstraps its PoW enclave after a restart.

Key enclave management. Instead of bootstrapping the global secret in entirety, SGXDedup generates the global secret in the key enclave based on two *sub-secrets* respectively owned by the cloud and the key server, so as to prevent either of them from learning the whole global secret.

To generate the global secret, we hard-code the cloud’s sub-secret into the key enclave code, and deliver the code (as a shared object) to the key server during SGXDedup’s initialization. We also implement a *secret generation ECall* for the key enclave, so as to allow the key server to provide its own sub-secret. The ECall can only be issued by the key server after the cloud’s sub-secret is included into the key enclave. It takes the key server’s sub-secret as its single input, and hashes the concatenation of the key server’s sub-secret and the cloud’s sub-secret to form the global secret. Note that the key server cannot access the enclave code and hence cannot learn the cloud’s sub-secret hard-coded inside the enclave (assuming that reverse engineering is impossible). Thus, even if the key server is compromised, the global secret remains secure, so the security of server-aided MLE is preserved. If both the key server and the cloud are simultaneously compromised, the security of SGXDedup reduces to that of the original MLE (§2.1).

PoW enclave management. When a client bootstraps its PoW enclave, it needs to attest the authenticity of the PoW enclave. However, remote attestation generally incurs a very large latency (e.g., about 9 s; see §5.1) to connect to the Intel service. Unlike the key enclave, whose remote attestation is only done once during initialization, the client needs to bootstrap and terminate the PoW enclave each time it joins and leaves SGXDedup, respectively. If remote attestation were used each time when the client joins, its substantial overhead will hurt usability.

SGXDedup leverages sealing to avoid remote attestation after the first bootstrap of the PoW enclave. Recall that the PoW enclave shares a PoW key with the cloud, such that the cloud can verify the authenticity of fingerprints (§3.1). Our idea is to seal the PoW key based on the measurement hash of the PoW enclave. Thus, when the client bootstraps again its PoW enclave, it unseals the PoW key into the bootstrapped PoW enclave. As long as the PoW key is recovered successfully, the authenticity of the bootstrapped PoW enclave is verified.

Specifically, the client first checks whether any sealed PoW key is locally available in its physical machine. If a sealed PoW key is not available (the first bootstrap), the client attests the PoW enclave via remote attestation and exchanges a PoW key with the cloud; otherwise if a sealed PoW key is available (after the first bootstrap), the client creates a new PoW enclave by loading the shared object, and calls the *key unsealing ECall* of the new PoW enclave to unseal the PoW key. The key unsealing ECall takes the address of the sealed PoW key as input. It derives a sealing key based on the measurement hash of the new PoW enclave, decrypts the sealed PoW key, and keeps it in the new PoW enclave.

When the client leaves SGXDedup, its PoW enclave needs to be terminated. The client issues the *key sealing ECall* to seal the PoW key. The key sealing ECall encrypts the PoW key based on the measurement hash of the PoW enclave, and stores the result in the address provided by the client.

3.3 Renewable Blinded Key Management

Each client securely communicates with the key enclave via a shared *blinded key* to prevent the eavesdropping by the key server (§3.1). To form the blinded key, a straightforward approach is to directly implement a key agreement protocol between the key enclave and each client. However, the key enclave needs to authenticate the client on-the-fly (e.g., a client may renew or revoke its cloud service subscription). Such dynamic authentication puts performance burdens on the key enclave.

SGXDedup manages blinded keys with the help of the cloud. During initialization, the cloud hard-codes a blinded secret κ into the key enclave code. Each client downloads a *key state* (derived from κ ; see below) from the cloud, and generates its blinded key (based on the key state) for the secure communication with the key enclave. Our rationales are two-fold. First, when the client issues any download request

from the cloud, the cloud can check if the client is authorized. Second, we can derive a sequence of renewable blinded keys from κ (instead of directly using κ), so as to prevent the revoked/compromised clients from persistently accessing the key enclave. As a side benefit, the renewable key management also prevents online brute-force attacks (§2.1) without actively slowing down the key generation rate [14].

SGXDedup uses *key regression* [30] to derive renewable blinded keys, while ensuring that the blinded keys in each client and the key enclave are consistent. Specifically, key regression works on a sequence of key states $S[1], S[2], \dots, S[m]$, each of which can be used to derive a key (e.g., via hashing). It allows the key enclave and the cloud to perform *rekeying* to derive a new state from an old state (e.g., deriving $S[2]$ from $S[1]$) using a *key regression secret*, such that the client cannot learn any information about new states without knowing the key regression secret. It also allows each client to derive any old state from the new state (e.g., deriving $S[1]$ from $S[2]$).

To realize key regression in SGXDedup, we use κ as the key regression secret shared between the cloud and the key enclave for deriving the new states and keys. In each upload, the client first downloads the up-to-date key state $S[i]$ from the cloud, and requests the current version number j of the blinded key accepted by the key enclave. Given that the key enclave may not renew the blinded key in time (e.g., it is busy with serving key generation exactly in the scheduled rekeying time), j is typically smaller than i (i.e., $S[j]$ is prior to $S[i]$). Then the client derives $S[j]$ from $S[i]$ and the corresponding blinded key $K[j]$, and communicates with the key enclave based on the same $K[j]$. Note that the cloud can derive $K[j]$, but it cannot eavesdrop the communication between each client and the key enclave, since the communication is additionally protected via the SSL/TLS-based channel between the client and the key server (see our assumptions in §2.3).

SGXDedup renews blinded keys in both the cloud and the key enclave. The cloud implements a timer to trigger rekeying over periodic time intervals. The key server issues the *rekeying ECall* to trigger rekeying when a scheduled rekeying time is reached.

Currently, we implement the hash-based key regression scheme [30] for high key derivation performance. Specifically, we define a parameter n (now set as 2^{20} [30]) to indicate the maximum number of affordable rekeying times. The cloud and the key enclave compute the i -th key state as $S[i] = \mathbf{H}^{n-i+1}(\kappa)$, and each client derives the corresponding blinded key as $K[i] = \mathbf{H}(S[i]||0^8)$, where $\mathbf{H}^{n-i+1}()$ iteratively calls the cryptographic hash function $\mathbf{H}()$ by $n-i+1$ times, and $||$ is the concatenation operator. To derive old states, the client downloads $S[i]$ and recovers $S[i-1] = \mathbf{H}(S[i])$.

3.4 SGX-Based Speculative Encryption

Given the shared blinded key (§3.3), the key enclave manages a secure channel with each client to protect the transferred fingerprints/keys during MLE key generation (§3.1). How-

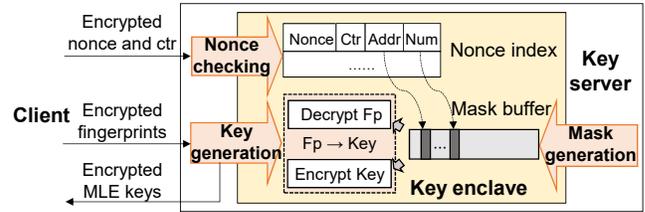


Figure 2: Overview of SGX-based speculative encryption.

ever, managing the secure channels, particularly with many clients, incurs high encryption/decryption costs in the key enclave. SGXDedup augments the secure communication channel management using *speculative encryption* [27] in the context of SGX, so as to offload the encryption/decryption overhead of the key enclave.

Speculative encryption. Speculative encryption [27] adopts encryption/decryption in *counter mode* [6] and pre-computes partial encryption/decryption results in an offline procedure, so as to reduce the online computational overhead in standalone cryptographic file systems. To encrypt a plaintext M , we first partition M into a sequence of plaintext blocks b_1, b_2, \dots, b_m (e.g., each block has a fixed size of 16 bytes). For each client, we pick a unique *nonce* θ that can be used only once by the encryption with the same key. We then compute the *mask* for the i -th plaintext block as $e_i = \mathbf{E}(K, \theta||i)$, where $\mathbf{E}()$ is a symmetric encryption function, K is the key, i is the *counter* (for counter-mode encryption), and $||$ is the concatenation operator. Finally, we compute each ciphertext block $c_i = e_i \oplus b_i$, where \oplus is the bitwise XOR operator, and form the whole ciphertext $C = c_1 || c_2 || \dots || c_m$. To decrypt the ciphertext, we generate the mask e_i for each block like above, recover the corresponding plaintext block $b_i = e_i \oplus c_i$ and hence the original plaintext M . Since the mask generation step is independent of each target plaintext/ciphertext, we can pre-compute the masks offline, followed by applying the lightweight XOR operation for online encryption/decryption.

Integration. To realize speculative encryption in SGXDedup, we need to address the nonce management issue. A unique nonce serves as an unpredictable “one-time-pad” that makes each counter-mode encryption output look random [6]. If counter-mode encryption is used by multiple clients, we need to associate each client with a unique nonce in its encryption/decryption operations. However, since different clients are isolated, it is challenging to ensure that the nonces associated with the clients are unique.

To ensure the uniqueness of a nonce, SGXDedup manages a centralized key-value store, called the *nonce index*, in the key enclave. Each entry of the nonce index maps a stored nonce (12 bytes) to three fields: its counter (4 bytes), the starting address (8 bytes) of the corresponding mask, and the number (4 bytes) of its available masks. Also, SGXDedup implements a *nonce checking ECall* (Figure 2) that can be called by the key server to compare the nonce submitted by

each client with the previously stored nonces in the nonce index and inform the client to re-pick a new one if a duplicate nonce is found. Note that the in-enclave nonce index can be effectively managed, since it can serve up to 112,000 clients (assuming one nonce per client) with only 3 MB EPC space (default configuration of SGXDedup).

SGXDedup applies speculative encryption to establish a secure channel between a client and the key enclave for the protection of MLE key generation. To initialize the secure channel, the client synchronizes its self-picked nonce θ and the corresponding counter i with the key enclave, where i is initialized as zero if θ has not been used for any encryption/decryption by the client. Specifically, it encrypts θ and i with the up-to-date blinded key (§3.3), computes a message authentication code (MAC) based on the resulting ciphertexts, and submits both the ciphertexts and the MAC to the key server. Here, we adopt *encrypt-then-MAC* [16] to detect any outdated blinded key by checking the MAC for any information transferred between the client and the key enclave.

The key server issues the nonce checking ECall, which takes the client's uploads as input, decrypts θ and i , and checks the decrypted θ and i with the nonce index:

- Case I: If θ is duplicate and $i = 0$, this indicates the re-use of an existing nonce, and the ECall returns a signal to inform the client to re-pick a new nonce.
- Case II: If θ is duplicate and $i \neq 0$, this implies that the nonce has been stored. The ECall updates the stored counter and marks the corresponding pre-computed masks (to be used for follow-up processing, as elaborated below).
- Case III: If θ is unique, this implies that the nonce is new, and the ECall adds θ into the nonce index.

For Cases II and III, the ECall accepts the communication and asks the client to transfer fingerprints for MLE key generation (§3.1). The client encrypts the fingerprints based on θ and i , and uploads the results; after encrypting each fingerprint block, the client increments i by one to prevent replay attacks. As shown in Figure 2, the key server issues the *key generation ECall* to process the encrypted fingerprints. The ECall checks if some masks are marked for the current client. If found (i.e., Case II), it uses the marked masks to decrypt the fingerprints and encrypt the generated MLE keys. Otherwise (i.e., Case III), it computes the masks online for decryption and encryption.

Mask pre-computation. To speed up encryption/decryption, the key enclave performs mask pre-computation when a new blinded key is applied (i.e., all existing masks are invalid) or a client has connected again after the last mask generation (i.e., some of its masks have been consumed). The key enclave calls the *mask generation ECall* (Figure 2), which pre-computes the masks for a number (e.g., three in our case) of most-recently-used nonces and writes the results into a *mask buffer*. By default, we configure the mask buffer with up to 90 MB. Suppose that each mask takes 16 bytes and the

average chunk size is 8 KB. The MLE key generation of a fingerprint consumes four masks: two are for decrypting the 32-byte fingerprint and the other two are for encrypting the resulting 32-byte MLE key. Thus, the pre-computed masks in the mask buffer can be used to process the fingerprints of up to 11.25 GB of data.

4 Implementation

We build an SGXDedup prototype in C++ using OpenSSL 1.1.1g [49], Intel SGX SDK 2.7 [3] and Intel SGX SSL [7]. Our prototype implements fingerprinting operations for plaintext and ciphertext chunks via SHA256, and chunk-based encryption via AES256. It contains about 14.2 K LoC.

Setup. To bootstrap the key enclave, the cloud hard-codes both the cloud's sub-secret (§3.2) and the blinded secret (§3.3) into the key enclave code. Alternatively, SGXDedup can also provision both secrets (using the secret provisioning functionality of SGX [3]) after authenticating the key enclave, at the expense of incurring extra bootstrapping overhead. The key enclave uses SHA256 to generate the global secret and implement the hash function in key regression. Each PoW enclave implements DHKE in NIST P-256 elliptic curve to share a PoW key with the cloud.

Key generation. Each client implements Rabin fingerprinting [52] for content-defined chunking. We fix the minimum, average, and maximum chunk sizes in Rabin fingerprinting at 4 KB, 8 KB, and 16 KB, respectively. To implement SGX-based speculative encryption (§3.4), we fix the nonce and the counter at 12 bytes and 4 bytes, respectively, and implement MACs using HMAC-SHA256. The key enclave generates the MLE key of each plaintext chunk via SHA256.

Deduplication. SGXDedup realizes source-based deduplication coupled with PoW. The PoW enclave implements a *proof generation ECall* to compute the fingerprints of ciphertext chunks and generate a signature based on the resulting fingerprints using AES-CMAC. The cloud implements the fingerprint index (§2.1) as a key-value store based on LevelDB [4]. To mitigate network and disk I/O costs, we store (non-duplicate) ciphertext chunks in 8 MB containers as units of transfers and storage [43].

Optimization. To reduce context switching and SGX memory encryption/decryption overhead, each client batches multiple fingerprints (4,096 by default) to transfer in the secure communication channel with the key enclave (§3.4). The key enclave processes each received batch of fingerprints. It accesses the batch via a pointer without copying the contents into the enclave [34]. Similarly, the PoW enclave processes the ciphertext chunks on a per-batch basis (4,096 by default) without content copy. We also use multi-threading to boost performance. Each client parallelizes the processing of chunking, fingerprinting of plaintext chunks, encryption, PoW, and uploads via multi-threading, while the key enclave and the cloud serve multiple connections in different threads.

5 Evaluation

We configure a LAN cluster of machines for multiple clients, the key server, and the cloud. Each machine has a quad-core 3.0 GHz Intel Core i5-7400 CPU, a 1 TB 7200 RPM SATA hard disk, and 8 GB RAM. All machines run Ubuntu 18.04 and are connected with 10 GbE. We evaluate SGXDedup using both synthetic (§5.1) and real-world (§5.2) workloads. We summarize the main results as follows.

- SGXDedup achieves high MLE key generation performance in both single-client (Exp#1) and multi-client (Exp#2) cases. For example, it achieves a $131.9\times$ speedup over OPRF-RSA adopted by DupLESS’s MLE key generation [14] in the single-client case.
- SGXDedup has $2.2\times$ and $8.2\times$ computational PoW speedups over universal hash-based PoW [60] (that only achieves weaker security) and Merkle-tree-based PoW [32] (Exp#3).
- SGXDedup has high overall performance in single-client (Exp#4 and Exp#5) and multi-client (Exp#6) cases. We also provide a time breakdown of SGXDedup in uploads (Exp#7). For example, in a 10 GbE LAN testbed, SGXDedup incurs only a slowdown of 17.5% in uploads compared to a plain deduplication system without any security protection; in real-cloud deployment (Exp#5), SGXDedup incurs a slowdown of 13.2% and its performance is bounded by the Internet bandwidth.
- SGXDedup is efficient for processing real-world workloads (Exp#8 and Exp#9). For example, its upload performance overhead over plain deduplication (without security protection) is within 22.0%; it also achieves high bandwidth savings over existing approaches [33, 42], by an absolute difference of up to 91.4%.

5.1 Evaluation on Synthetic Workloads

We generate a synthetic dataset with a set of files, each of which comprises globally non-duplicate chunks. By default, we set the file size as 2 GB (except for Exp#2, in which we stress-test the MLE key generation performance). A client uploads or downloads a file via the cloud. To avoid the performance impact of disk I/O, we store the file data in memory rather than on disk (except for Exp#5, in which we process on-disk files in real-cloud deployment). We plot the average results over 10 runs. We also include the 95% confidence intervals from *Student’s t-Distribution* into bar charts (for brevity, we exclude them from line charts).

Exp#1 (Single-client MLE key generation). We evaluate MLE key generation in two rounds. First, a client creates the plaintext chunks of a 2 GB file via Rabin fingerprinting (§4) and issues MLE key generation requests. It then repeats the MLE key generation process for the chunks of a different 2 GB file. The difference is that the second round uses the pre-computed masks for MLE key generation (§3.4).

We compare the single-client MLE key generation speed of SGXDedup with state-of-the-arts. We consider two OPRF-

based key generation approaches, namely *OPRF-BLS* [10] and *OPRF-RSA* [14], which implement the OPRF primitive (§2.1) based on blind BLS and blind RSA signatures, respectively. We also consider two relaxed key generation approaches, namely *MinHash encryption* [51] and *TED* [41], which trade storage efficiency and security for performance. Specifically, MinHash encryption generates MLE keys using OPRF-RSA on a per-segment basis, where the average segment size is configured as 1 MB. TED generates the MLE key for each chunk based on the sketch-based frequency counting of the short hashes of the chunk (c.f. §3.3 in [41]).

Figure 3 shows the results. SGXDedup outperforms all baseline approaches, by avoiding the expensive cryptographic primitives in OPRF-BLS, OPRF-RSA, and MinHash encryption and the frequency counting computations in TED. Its first round achieves $1,583\times$ and $131.9\times$ speedups over OPRF-BLS and OPRF-RSA, respectively. The speedups are $9.4\times$ and $3.7\times$ over MinHash encryption and TED, respectively, even though the latter two sacrifice storage efficiency and security. Speculative encryption in the second round improves the MLE key generation speed of the first round by 67.8%.

Exp#2 (Multi-client MLE key generation). We evaluate multi-client MLE key generation. For stress-testing, we configure a single machine that runs multiple threads, each of which simulates a client, to simultaneously issue MLE key generation requests to the key server (which runs on a different machine). Recall that the pre-computed masks in the key enclave can be used to efficiently process at most 11.25 GB of data (§3.4). To enable speculative encryption for each simulated client in the second round, we configure each thread to generate 40,960 fingerprints (i.e., 320 MB of raw data for 8 KB chunks) and configure the key enclave to equally pre-compute masks for each simulated client after the first round of key generation. We measure the *aggregate* MLE key generation speed on processing all MLE key generation requests from all simulated clients.

Figure 4 shows the results. The aggregate key generation speeds of both rounds initially increase with the number of simulated clients. At peak, the first and second rounds achieve 8.5×10^5 keys/s and 29×10^5 keys/s for five and ten simulated clients, respectively. After ten clients, the aggregate speeds drop due to aggravated context switching overhead. On average, speculative encryption in the second round achieves $4.4\times$ aggregate key generation speedup over the first round.

Exp#3 (Computational PoW). We evaluate the PoW performance. We consider a single client that performs PoW on a 2 GB file. The client creates plaintext chunks from the file, encrypts each plaintext chunk, and issues PoW requests to the cloud. We measure the *computational PoW speed* based on the total computational time of all chunks in both the client (where the PoW enclave performs fingerprinting on the ciphertext chunks and signs the resulting fingerprints) and the cloud (which verifies the authenticity of the fingerprints); note that we exclude the network transfer time between the

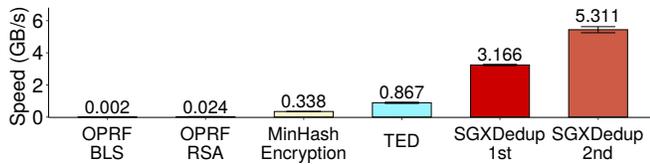


Figure 3: (Exp#1) Single-client MLE key generation.

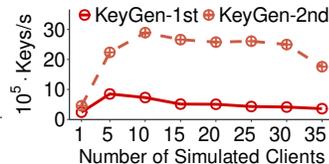


Figure 4: (Exp#2) Multi-client MLE key generation.

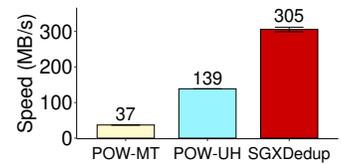


Figure 5: (Exp#3) Computational PoW.

client and the cloud in our speed calculation (we consider the network transfer time in Exp#11 in Appendix).

We compare SGXDedup with two state-of-the-art PoW approaches: (i) *PoW-MT* [32] (a.k.a. the basic version in [32]), a Merkle-tree-based PoW approach that encodes the chunks with erasure coding and builds a Merkle tree over the encoded content for PoW; and (ii) *PoW-UH* [60], which builds on universal hashing but trades security for performance. For fair comparisons, we implement both PoW-MT and PoW-UH in C++ by ourselves. Note that there are improved PoW approaches [32], but they incur even higher performance overhead for low memory usage.

Figure 5 shows the results. SGXDedup dramatically outperforms PoW-MT, since it avoids erasure coding and Merkle tree construction in the client, as well as Merkle-tree-based verification in the cloud. It achieves an 8.2× speedup over PoW-MT. It also achieves a 2.2× speedup over PoW-UH.

Exp#4 (Single-client uploads and downloads). We consider a single client, and compare the upload and download performance of SGXDedup with two baseline systems: (i) *PlainDedup*, which disables the key generation, encryption and PoW operations of SGXDedup, and hence realizes source-based deduplication without any security protection; and (ii) *DupLESS* [14], which generates per-chunk MLE keys via OPRF-RSA, performs encryption and uploads all ciphertext chunks to the cloud for deduplication. Since the original implementation of DupLESS does not provide a deduplicated storage backend (it assumes that Dropbox is used), we implement DupLESS in C++ based on the design described in [14] by ourselves. Note that PlainDedup retrieves files based on file recipes that are not encrypted, and differs from the two-round downloads in encrypted deduplication systems (i.e., both SGXDedup and DupLESS); in the latter, the client first downloads and decrypts the file recipe, followed by downloading the chunks to reconstruct the file (§2.1).

We evaluate the upload and download speeds in three steps: (i) a client first uploads a 2 GB file; (ii) the client restarts and then uploads another 2 GB file that is identical to the previous one; and (iii) the client downloads the file. Note that the second upload performs source-based deduplication (for both PlainDedup and SGXDedup), and leverages the pre-computed masks to accelerate key generation (only for SGXDedup).

Figure 6(a) shows the upload speeds for different network bandwidths controlled by *trickle* [28]. For the first upload, when the network bandwidth is 1 Gbps, the upload speeds of both SGXDedup (106.6 MB/s) and PlainDedup (106.2 MB/s)

are bound by the network speed, while the performance bottleneck of DupLESS (20.1 MB/s) is the OPRF-RSA-based key generation (Exp#1). When the network bandwidth increases to 10 Gbps (the default), the upload speeds of SGXDedup and PlainDedup achieve 193.6 MB/s and 242.0 MB/s, respectively, while that of DupLESS keeps stable at 20.0 MB/s. For the second upload, DupLESS achieves the same speed as the first upload due to its key generation performance bottleneck. The upload speeds of both SGXDedup and PlainDedup are less influenced by the network bandwidth since they do not need to transfer data. On average, SGXDedup achieves 8.1× and 9.6× speedups over DupLESS in the first and the second uploads, respectively. Even compared with the insecure PlainDedup, SGXDedup only incurs about 17.5% and 21.4% drops of the corresponding upload speeds. The overhead comes from the security mechanisms of SGXDedup, including key generation, encryption, and PoW.

Figure 6(b) shows the download speeds. As the network bandwidth increases to 10 Gbps, both SGXDedup and DupLESS achieve 323.1 MB/s, a 44.2% drop from PlainDedup. The reason is that they serially retrieves and decrypts the file recipe, followed by downloading the ciphertext chunks.

Exp#5 (Real-cloud uploads and downloads). We now extend Exp#4 and evaluate the upload and download speeds in a real-cloud deployment. Specifically, we deploy the client and the key server in our LAN testbed (§5), and connect the client via the Internet to the *Alibaba Cloud*, in which we rent a virtual machine *ecs.c6e.xlarge* to run the cloud. The cloud machine is equipped with a quad-core 3.2 GHz CPU (Intel Xeon Cascade Lake Platinum 8269CY in its host platform), 8 GB memory. We mount the cloud with *Alibaba General Purpose NAS* as the storage backend. The NAS can achieve up to 20000 IOPS for 4 K random reads and writes.

We use on-disk data files for uploads (as opposed to Exp#4, which loads files into client’s memory before uploads), and allow the cloud to store the received data files in NAS. We also use *scp* to upload the whole data file (i.e., 2 GB) from the client to the cloud, so as to provide a data transfer benchmark in the Internet environment.

Table 2 shows the results. In the first upload, the performance of all systems (11.4 MB/s for SGXDedup, 11.6 MB/s for PlainDedup and 10.8 MB/s for DupLESS) is bounded by Internet bandwidth (11.9 MB/s). In the second upload, SGXDedup achieves 104.3 MB/s, 9.7× speedup over DupLESS and 13.2% drop compared to PlainDedup. Note that the performance differences are smaller than those in Exp#4, since

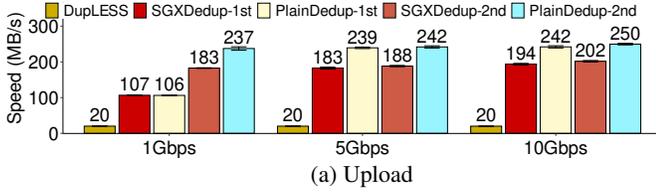


Figure 6: (Exp#4) Single-client uploads and downloads. We exclude the second upload speed of DupLESS (which has the same performance in two uploads) and the download speed of DupLESS (which is identical to that of SGXDedup).

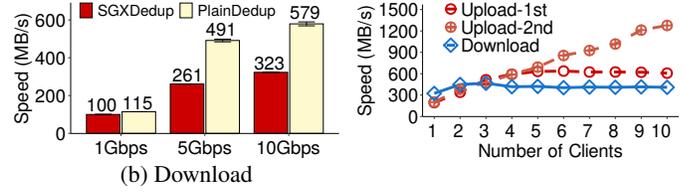


Figure 7: (Exp#6) Multi-client uploads and downloads.

Approach	First Upload	Second Upload	Download
Transfer	11.9 ± 0.03		
SGXDedup	11.4 ± 0.3	104.3 ± 1.2	10.1 ± 0.1
PlainDedup	11.6 ± 0.1	120.1 ± 1.4	11.3 ± 0.3
DupLESS	10.8 ± 0.2		10.1 ± 0.1

Table 2: (Exp#5) Real-cloud upload and download (unit: MB/s).

SGXDedup and PlainDedup are now bounded by client-side disk I/Os. In download, the performance of all three systems is bounded by Internet bandwidth again, while SGXDedup and DupLESS degrade the download speed of PlainDedup by 10.6% due to their serial retrieval and decryption (Exp#4).

Exp#6 (Multi-client uploads and downloads). We now consider multiple clients that issue uploads/downloads concurrently. We focus on SGXDedup, and configure the key enclave to equally pre-compute masks for each client after the first upload. We fix the network bandwidth at 10 Gbps, and evaluate the *aggregate* upload and download speeds for all clients to complete the uploads/downloads.

Figure 7 shows the results with up to ten clients. The aggregate upload speed in the second round increases with the number of clients, and reaches 1277.1 MB/s. On the other hand, the aggregate upload speed in the first round increases to 637.0 MB/s for seven clients, followed by dropping to 620.3 MB/s for ten clients due to the write contention across clients. Similarly, the aggregate download speed finally drops to 408.8 MB/s for the read contention of multiple clients.

Exp#7 (Time breakdown). We present a time breakdown of SGXDedup to study the performance of different steps. Suppose that the key enclave has been started and we focus on the initialization and the upload procedures of a single client. The initialization procedure bootstraps the PoW enclave of the client. The upload procedure includes the following steps: (i) *chunking*, which partitions an input file into plaintext chunks; (ii) *fingerprinting-p*, which computes the fingerprints of plaintext chunks; (iii) *key generation*, which generates MLE keys from the key enclave; (iv) *encryption*, which encrypts the plaintext chunks; (v) *fingerprinting-c*, in which the PoW enclave computes the fingerprints of ciphertext chunks; (vi) *signing*, in which the PoW enclave computes the signature of the fingerprints; (vii) *verification*, in which the cloud verifies the authenticity of received fingerprints; (viii) *deduplication*, in which the cloud detects duplicate ciphertext chunks and informs the client; and (ix) *transfer*, which uploads the non-

Procedure/Step	First Upload	Second Upload
Initialization	9.38 ± 2.72 s	0.80 ± 0.004 s
Chunking	3.77 ± 0.15 ms	
Fingerprinting-p	3.24 ± 0.28 ms	
Key generation	0.31 ± 0.01 ms	0.18 ± 0.01 ms
Encryption	2.47 ± 0.10 ms	
PoW	Fingerprinting-c	3.28 ± 0.01 ms
	Signing	0.01 ± 0.00004 ms
	Verification	0.005 ± 0.00003 ms
Deduplication	0.38 ± 0.03 ms	0.48 ± 0.03 ms
Transfer	1.29 ± 0.09 ms	0.05 ± 0.01 ms

Table 3: (Exp#7) Time breakdown per 1 MB of file data processed: fingerprinting-p and fingerprinting-c are operated on plaintext and ciphertext chunks, respectively.

duplicate ciphertext chunks and the file recipe.

Table 3 presents the results (per 1 MB of file data processed). The initialization procedure is time-consuming in the first upload since it needs to contact the Intel attestation service for checking the integrity of the PoW enclave. When the client restarts again, it no longer needs to execute remote attestation and reduces the setup time of the first round by 91.5%. Note that the time overhead of the initialization procedure can be amortized across multiple uploads and downloads.

The key generation step is efficient, and takes up to 2.1% of the overall upload time. With speculative encryption, SGXDedup further reduces the key generation time of the first upload by 41.9%. Also, the overall PoW step takes up to 24.4% of the overall upload time, and the dominant computation in PoW is the fingerprinting of ciphertext chunks, which is necessary for finding duplicates in encrypted deduplication. With the lightweight signing and verification steps (that take up to 0.1% of the overall upload time), we protect source-based deduplication against side-channel attacks, while reducing the transfer time of the first upload by 96.1%.

Additional experiments. In our technical report [53], we study the impact of batch size on key generation and PoW performance, as well as the rekeying latency.

5.2 Evaluation on Real-world Workloads

We evaluate SGXDedup using real-world workloads (which contain duplicates). We consider two real-world datasets in our evaluation.

- The first dataset, called *FSL*, contains the backup snap-

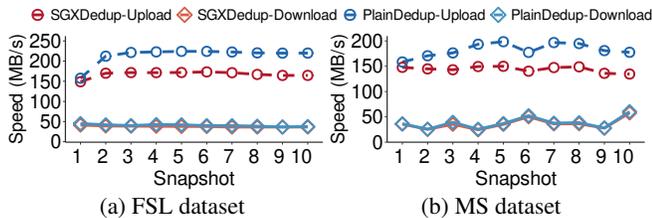


Figure 8: (Exp#8) Trace-driven upload and download performance.

shots of user home directories from a shared file system at the File systems and Storage Lab (FSL) [5, 56]. Each snapshot lists the 48-bit fingerprints of the chunks of an average chunk size of 8 KB. We select all snapshots (under `fs1homes`) from January 22 to June 17, 2013, covering a total of 56.2 TB of pre-deduplicated data. The dataset reduces to 431.9 GB after deduplication (i.e., a 133.2× deduplication ratio).

- The second dataset, called *MS*, contains the Windows file system snapshots from Microsoft [45]. Each snapshot lists the 40-bit fingerprints of the chunks of an average chunk size of 8 KB. We sample 140 snapshots from the original 857 snapshots, such that each snapshot has a size of about 100 GB. Our dataset contains 14.4 TB of pre-deduplicated data. It reduces to 2.4 TB after deduplication (i.e., a 6× deduplication ratio).

Exp#8 (Trace-driven performance). We conduct trace-driven evaluation on the upload and download performance of SGXDedup. We choose ten snapshots from FSL and MS each as follows. For FSL, we pick the snapshots from the same user to have high cross-snapshot redundancies; for MS, we pick the snapshots that have the most intra-snapshot redundancies. The chosen FSL and MS snapshots take 1.3 TB and 1.0 TB of pre-deduplicated data, respectively. Since our snapshots only contain fingerprints without actual data, we reconstruct each plaintext chunk by repeatedly writing its fingerprints into a spare chunk until reaching the specified chunk size, so the same (distinct) fingerprint returns the same (distinct) chunk. We use a single client to upload the snapshots one by one, followed by downloading them in the same order of uploads. We compare SGXDedup with PlainDedup (Exp#4).

Figure 8(a) shows the upload and download speeds across FSL snapshots. Both SGXDedup and PlainDedup achieve high upload speeds, especially when uploading the subsequent snapshots (e.g., at least 164.5 MB/s for SGXDedup and 212.2 MB/s for PlainDedup) after the first snapshot (e.g., 148.8 MB/s for SGXDedup and 157.5 MB/s for PlainDedup). The reason is that the FSL dataset has high redundancies across snapshots, and both systems can upload less data for subsequent snapshots. On average, SGXDedup incurs an upload performance drop of 22.0% compared to PlainDedup. Note that the overhead is slightly larger than that (17.5-21.4%) in our performance evaluation (Exp#4) using synthetic workloads. The reason is that chunking is now

disabled in trace-driven evaluation, and the bottleneck for SGXDedup switches to PoW (see Table 3), while the bottleneck for PlainDedup is fingerprinting. The download speed decreases from 41.3 MB/s to 36.4 MB/s for SGXDedup, and from 45.0 MB/s to 37.9 MB/s for PlainDedup, mainly due to chunk fragmentation [43]. We can mitigate chunk fragmentation via re-writing and caching [19, 43].

Figure 8(b) shows the upload and download speeds across MS snapshots. Both systems have lower upload speeds than in the FSL dataset, since the MS dataset has more non-duplicate chunks (e.g., 28.7 M for MS versus 18.3 M for FSL) and aggravates the access overhead of the fingerprint index. On average, SGXDedup incurs an upload slowdown of 21% compared to PlainDedup. Note that the download speeds fluctuate (e.g., 24.3-57.5 MB/s for SGXDedup and 25.6-60.3 MB/s for PlainDedup), since some MS snapshots have more non-duplicate chunks and are likely to be stored in consecutive regions that can be accessed quickly via sequential reads (i.e., less chunk fragmentation [43]).

Exp#9 (Bandwidth savings). We evaluate the bandwidth efficiency of SGXDedup. We consider two existing approaches that defend against side-channel attacks using both source-based deduplication (i.e., the client performs deduplication and uploads only non-duplicate ciphertext chunks to the cloud) and target-based deduplication (i.e., the client uploads all ciphertext chunks to the cloud, which performs deduplication on the received ciphertext chunks): (i) *two-stage deduplication* [42], which applies source-based deduplication on individual users, followed by target-based deduplication across users; and (ii) *randomized-threshold deduplication* [33], which performs either source-based deduplication or target-based deduplication based on a randomly chosen threshold. We choose the upper and lower bounds of the threshold in randomized-threshold deduplication as 20 and 2, respectively, as in [33]. For FSL, we aggregate the same-day snapshots of different users, and add each aggregate snapshot into the storage in the order of its creation time. For MS, we add each snapshot based on its snapshot ID (we assume that each MS snapshot is from a distinct user). We measure the *bandwidth savings* as the fraction of data reduction in transmission compared to the pre-deduplicated data. Here, we do not consider the bandwidth overhead due to metadata.

Figure 9 shows the cumulative bandwidth saving after uploading each snapshot. After uploading all snapshots, SGXDedup achieves 99.2% and 83.2% of bandwidth savings in FSL and MS, respectively. Since SGXDedup performs source-based deduplication, its bandwidth savings are also translated to the storage savings. Two-stage deduplication has almost identical bandwidth savings to SGXDedup in FSL, since the FSL dataset includes a large volume of intra-user redundancies. On the other hand, in MS, two-stage deduplication only has 47.9% of bandwidth savings (less than SGXDedup by an absolute difference of 35.3%). Randomized-threshold deduplication has varying bandwidth savings, with 48.5% in MS

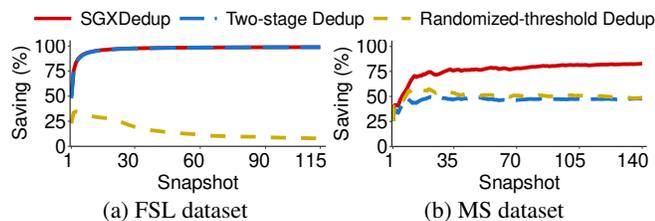


Figure 9: (Exp#9) Cumulative bandwidth savings after each snapshot is stored.

but only 7.8% in FSL (less than SGXDedup by an absolute difference of 34.7% and 91.4%, respectively).

6 Related Work

MLE key management. Traditional MLE-based [15] encrypted deduplication systems (e.g., [8, 20, 55]) are vulnerable to offline brute-force attacks [14]. DupLESS [14] proposes server-aided MLE to perform MLE key generation in a dedicated key server. Follow-up studies on server-aided MLE focus on deduplication pattern attestation [10], cross-user deduplication [62], and MLE key renewal [51].

Some studies mitigate the overhead of chunk-based MLE key generation at the expense of degrading deduplication effectiveness [51, 62] or weakening security [40]. SGXDedup outperforms these approaches in performance, while preserving both deduplication effectiveness and security (§5.1). Some other MLE key generation approaches include threshold-based key management [26] and decentralized key management [44], but they build on the cryptographic primitives (e.g., threshold signature [26] and password-authenticated key exchange [44]) that are theoretically proven but not readily implemented.

Defenses against side-channel attacks. Source-based deduplication is bandwidth-efficient but vulnerable to side-channel attacks [33]. Prior studies [33, 42] combine source-based deduplication and target-based deduplication to defend against side-channel attacks, while SGXDedup achieves significantly more bandwidth savings by purely performing source-based deduplication (§5.2) and using PoW to protect against side-channel attacks. Also, SGXDedup is much more efficient than Merkle-tree-based PoW (§5.1). Some other studies make PoW efficient by relaxing security (e.g., [23, 60]), while SGXDedup uses client-side SGX to preserve the security of PoW.

SGX-based storage. SGX [3] has been widely used for securing storage systems. PESOS [39] enforces the access policies of object storage with SGX. OBLIVIATE [9] enhances the security of SGX-based file systems against privileged side-channel attacks. EnclaveDB [50] and OblIDB [29] protect outsourced databases against information leakage via SGX. NEXUS [24] enables fine-grained access control with SGX over untrusted cloud storage. On the performance side, Harnik *et al.* [34] propose guidelines on mitigating the perfor-

mance overhead of SGX implementations. ShieldStore [37] implements application-specific data management to limit the enclave memory usage. SPEICHER [12] is an SGX-based LSM-based key-value store with efficient I/O operations.

All the above studies do not consider deduplication. Dang *et al.* [22] propose proxy-based protocols for bandwidth-efficient encrypted deduplication, but the protocols do not address the key generation performance overhead and have no implementation. SPEED [21] leverages deduplication to make SGX computations efficient, but SGXDedup improves the performance of encrypted deduplication with SGX. Other studies use a cloud-side enclave for PoW verification [61] and secure file-based deduplication [31], while SGXDedup uses a client-side enclave for efficient PoW proof generation and supports more fine-grained chunk-based deduplication.

7 Conclusion

This paper addresses the performance overhead of encrypted deduplication via SGX. We present SGXDedup, which implements a set of ECalls to run sensitive operations in SGX enclaves, so as to accelerate encrypted deduplication while preserving security. SGXDedup incorporates three key designs: (i) the secure and efficient enclave management, (ii) the renewable blinded key management, and (iii) the design of SGX-based speculative encryption for lightweight computations. Evaluation on our SGXDedup prototype demonstrates its high performance in synthetic and real-world workloads.

Acknowledgments

We thank our shepherd, Russell Sears, and the anonymous reviewers for their valuable comments. We thank Changchun Li for his help in the prototype evaluation, and Cheng Li for his feedbacks on the paper draft. This work was supported in part by grants by the National Natural Science Foundation of China (61972073), the Key Research Funds of Sichuan Province (2020YFG0298, 2021YFG0167), Sichuan Science and Technology Program (2020JDTD0007), the Fundamental Research Funds for Chinese Central Universities (ZYGX2020ZB027), Innovation and Technology Fund (ITS/315/18FX), and CUHK Direct Grant 2020/21 (4055148).

References

- [1] AMD secure encrypted virtualization (SEV). <https://developer.amd.com/sev/>.
- [2] Intel architecture memory encryption technologies specification. <https://software.intel.com/sites/default/files/managed/a5/16/Multi-Key-Total-Memory-Encryption-Spec.pdf?source=techstories.org>.
- [3] Intel software guard extensions SDK. <https://software.intel.com/en-us/sgx/sdk>.
- [4] LevelDB. <https://github.com/google/leveldb>.

- [5] Traces and snapshots public archive. <http://tracer.filesystems.org>.
- [6] Using advanced encryption standard (AES) counter mode with ipsec encapsulating security payload (ESP). <https://tools.ietf.org/html/rfc3686>.
- [7] Intel software guard extensions SSL. <https://github.com/intel/intel-sgx-ssl>, 2017.
- [8] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of ACM OSDI*, 2002.
- [9] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. Obliviate: A data oblivious file system for Intel SGX. In *Proc. of NDSS*, 2018.
- [10] F. Armknecht, J.-M. Bohli, G. O. Karame, and F. Youssef. Transparent data deduplication in the cloud. In *Proc. of ACM CCS*, 2015.
- [11] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proc. of ACM CCS*, 2007.
- [12] M. Baillieu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. SPEICHER: Securing LSM-based key-value stores using shielded execution. In *Proc. of USENIX FAST*, 2019.
- [13] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. In *Proc. of USENIX OSDI*, 2014.
- [14] M. Bellare, S. Keelveedhi, and T. Ristenpart. DupLESS: server-aided encryption for deduplicated storage. In *Proc. of USENIX Security*, 2013.
- [15] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. In *Proc. of EuroCrypt*, 2013.
- [16] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Proc. of AsiaCrypt*, 2000.
- [17] J. Black. Compare-by-hash: A reasoned analysis. In *Proc. of USENIX ATC*, 2006.
- [18] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proc. of USENIX Security*, 2018.
- [19] Z. Cao, H. Wen, F. Wu, and D. H. Du. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *Proc. of USENIX FAST*, 2018.
- [20] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proc. of ACM OSDI*, 2002.
- [21] H. Cui, H. Duan, Z. Qin, C. Wang, and Y. Zhou. SPEED: Accelerating enclave applications via secure deduplication. In *Proc. of ICDCS*, 2019.
- [22] H. Dang and E.-C. Chang. Privacy-preserving data deduplication on trusted processors. In *Proc. of IEEE CLOUD*, 2017.
- [23] R. Di Pietro and A. Sorniotti. Boosting efficiency and security in proof of ownership for deduplication. In *Proc. of ACM ASIACCS*, 2012.
- [24] J. B. Djoko, J. Lange, and A. J. Lee. NEXUS: Practical and secure access control on untrusted storage platforms using client-side SGX. In *Proc. of IEEE/IFIP DSN*, 2019.
- [25] J. R. Douceur, A. Adya, W. J. Bolosky, P. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proc. of IEEE ICDCS*, 2002.
- [26] Y. Duan. Distributed key generation for encrypted deduplication: Achieving the strongest privacy. In *Proc. of ACM CCSW*, 2014.
- [27] V. Eduardo, L. C. E. de Bona, and W. M. N. Zola. Speculative encryption on GPU applied to cryptographic file systems. In *Proc. of USENIX FAST*, 2019.
- [28] M. A. Eriksen. Trickle: A userland bandwidth shaper for UNIX-like systems. In *Proc. of USENIX ATC*, 2005.
- [29] S. Eskandarian and M. Zaharia. OblidB: Oblivious query processing for secure databases. In *Proc. of ACM VLDB*, 2019.
- [30] K. Fu, S. Kamara, and T. Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *Proc. of NDSS*, 2006.
- [31] B. Fuhry, L. Hirschhoff, S. Koesnadi, and F. Kerschbaum. SeGShare: Secure group file sharing in the cloud using enclaves. In *Proc. of IEEE/IFIP DSN*, 2020.
- [32] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg. Proofs of ownership in remote storage systems. In *Proc. of ACM CCS*, 2011.
- [33] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, 2010.
- [34] D. Harnik, E. Tsfadia, D. Chen, and R. Kat. Securing the storage data path with SGX enclaves. <https://arxiv.org/abs/1806.10883>, 2018.
- [35] R. Hasan, W. Yurcik, and S. Myagmar. The evolution of storage service providers: techniques and challenges to outsourcing storage. In *Proc. of ACM StorageSS*, 2005.

- [36] A. Juels and B. S. Kaliski, Jr. PORs: Proofs of retrievability for large files. In *Proc. of ACM CCS*, 2007.
- [37] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh. ShieldStore: Shielded in-memory key-value storage with SGX. In *Proc. of ACM Eurosys*, 2019.
- [38] R. Kotla, L. Alvisi, and M. Dahlin. SafeStore: A durable and practical storage system. In *Proc. of USENIX ATC*, 2007.
- [39] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer. PESOS: Policy enhanced secure object store. In *Proc. of ACM EuroSys*, 2018.
- [40] J. Li, P. Lee, C. Tan, C. Qin, and X. Zhang. Information leakage in encrypted deduplication via frequency analysis: Attacks and defenses. *ACM Transactions on Storage*, 16(1):4:1–4:30, 2020.
- [41] J. Li, Z. Yang, Y. Ren, P. Lee, and X. Zhang. Balancing storage efficiency and data confidentiality with tunable encrypted deduplication. In *Proc. of ACM Eurosys*, 2020.
- [42] M. Li, C. Qin, and P. Lee. CDStore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal. In *Proc. of USENIX ATC*, 2015.
- [43] M. Lillibridge, K. Eshghi, and D. Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proc. of USENIX FAST*, 2013.
- [44] J. Liu, N. Asokan, and B. Pinkas. Secure deduplication of encrypted data without additional independent servers. In *Proc. of ACM CCS*, 2015.
- [45] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *Proc. of USENIX FAST*, 2011.
- [46] S. Mofrad, F. Zhang, S. Lu, and W. Shi. A comparison study of Intel SGX and AMD memory encryption technology. In *Proc. of ACM HASP*, 2018.
- [47] M. Naor and O. Reingold. Number-theoretic constructions of efficient pseudo-random functions. *Journal of the ACM*, 51(2):231–262, 2004.
- [48] O. Oleksenko, B. Trach, R. Krahn, A. Martin, C. Fetzer, and M. Silberstein. Varys: Protecting SGX enclaves from practical side-channel attacks. In *Proc. of USENIX ATC*, 2018.
- [49] OpenSSL. Cryptography and SSL/TLS toolkit. <https://www.openssl.org/>.
- [50] C. Priebe, K. Vaswani, and M. Costa. EnclaveDB: A secure database using SGX. In *Proc. of IEEE S&P*, 2018.
- [51] C. Qin, J. Li, and P. Lee. The design and implementation of a rekeying-aware encrypted deduplication storage system. *ACM Transactions on Storage*, 13(1):9:1–9:30, 2017.
- [52] M. O. Rabin. Fingerprint by random polynomials. Technical report.
- [53] Y. Ren, J. Li, Z. Yang, P. P. C. Lee, and X. Zhang. Accelerating encrypted deduplication via SGX. Technical report, CUHK, 2021. http://www.cse.cuhk.edu.hk/~pclee/www/pubs/tech_sgxdedup.pdf.
- [54] H. Ritzdorf, G. O. Karame, C. Soriente, and S. Čapkun. On information leakage in deduplicated storage systems. In *Proc. of ACM CCSW*, 2016.
- [55] P. Shah and W. So. Lamassu: Storage-efficient host-side encryption. In *Proc. of USENIX ATC*, 2015.
- [56] Z. Sun, N. Xiao, G. Kuenning, S. Mandal, E. Zadok, P. Shilane, and V. Tarasov. A long term user-centric analysis of deduplication patterns. In *Proc. of IEEE MSST*, 2015.
- [57] A. S. Technology. Building a secure system using TrustZone® technology. https://static.docs.arm.com/gencc009492/c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.
- [58] M. Vrable, S. Savage, and G. M. Voelker. Cumulus: Filesystem backup to the cloud. In *Proc. of USENIX FAST*, 2009.
- [59] G. Wallace, F. Douglass, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *Proc. of USENIX FAST*, 2012.
- [60] J. Xu, E.-C. Chang, and J. Zhou. Weak leakage-resilient client-side deduplication of encrypted data in cloud storage. In *Proc. of ACM ASIACCS*, 2013.
- [61] W. You and B. Chen. Proofs of ownership on encrypted cloud data via Intel SGX. In *Proc. of ACNS*, 2020.
- [62] Y. Zhou, D. Feng, W. Xia, M. Fu, F. Huang, Y. Zhang, and C. Li. SecDep: A user-aware efficient fine-grained secure deduplication scheme with multi-level key management. In *Proc. of IEEE MSST*, 2015.
- [63] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. of USENIX FAST*, 2008.
- [64] P. Zuo, Y. Hua, C. Wang, W. Xia, S. Cao, Y. Zhou, and Y. Sun. Mitigating traffic-based side channel attacks in bandwidth-efficient cloud storage. In *Proc. of IEEE IPDPS*, 2018.

ICARUS: Attacking low Earth orbit satellite networks

Giacomo Giuliani, Tommaso Ciussani, Adrian Perrig, Ankit Singla
Department of Computer Science, ETH Zürich

Abstract

Internet service based on low Earth orbit satellites is generating immense excitement in the networking community due to its potential for global low-latency connectivity. Despite the promise of LEO satellite networks, the security of their operation has so far been largely neglected. In this context, we present ICARUS, a new class of denial of service attacks on LEO networks. ICARUS turns these networks' key benefits into vulnerabilities: an adversary can leverage the direct global accessibility to launch an attack from numerous locations, while the quest for low latency constrains routing, and provides predictability to the adversary. We explore how the adversary can exploit other unique features, including the path structure of such networks, and the public knowledge of the locations and connectivity of the satellite-routers. We find that a small amount of attack bandwidth can hamper communications between large terrestrial areas. Finally, we lay out open problems in this direction, and provide a framework to enable further research on attacks and defenses in this context.

1 Introduction

SpaceX Starlink [60,61], Amazon Kuiper [35], and others are deploying hundreds to thousands of satellites, each carrying high-capacity networking equipment. These low Earth orbit (LEO) satellite networks (LSNs) aim to provide global broadband Internet service. While global access would itself be a huge leap in connectivity, these networks also promise low-latency communications between otherwise well-connected regions. For instance, recent work [9] shows that for long-distance communication beyond a few thousand kilometers, such networks could provide lower latency than not only today's fiber Internet, but also specialized free-space radio networking used in the high-frequency trading industry.

Driven by the exciting potential of global low-latency networking, researchers are exploring many interesting problems related to LSNs, such as estimating their latency improvements [9, 28], and network topology design [10] and new routing strategies [25, 29, 33].

However, prior work has thus far largely ignored the aspect of the secure operation of these networks in the face of adversaries. We hence explore the resilience of LSNs to a large-scale volumetric distributed denial-of-service (DDoS) attack, carried out by a botnet of compromised satellite-enabled hosts. This threat has been extensively studied in terrestrial networks, but as we detail later (§ 3.4), an LSN is an extremely different environment: an adversary can leverage its global footprint to flexibly inject traffic into the network; knowledge about node (satellite) locations and network structure, far from be-

ing closely guarded, is easily available public information; and the low-latency objective both limits path diversity and reduces uncertainty about routing for the adversary.

With these unique opportunities for the adversary, also come new challenges: unlike a terrestrial transit network, the LSN connects directly to the clients injecting traffic into it, and can more easily detect and stop malicious behavior; at least initially, the client population will be much smaller than the Internet, limiting the adversary's resources; and the sparse structure of the inter-satellite network creates a risk for attack traffic flows to congest each other instead of the target(s).

Leveraging the unique opportunities and addressing the new challenges, we present ICARUS, a DDoS attack on LSNs. ICARUS draws on Coremelt [64] and Crossfire [31], and other DDoS work, but is customized for LSNs — the adversary carefully plans attack traffic using an LSN's structure, with the objective of congesting a target link or set of target links. The adversary hopes to do so at low *cost*, in terms of attack traffic volume, and with low *detectability*, in terms of change in the ingress bandwidth at individual satellites.

To help develop intuition in this new setting, we first analyze ICARUS in the scenario where the LSN uses single shortest-path routing. By simulating the attack on the LSN with the largest deployment to date, Starlink, we find that the adversary can successfully target the majority of its links, congesting both ground-to-satellite links and inter-satellite links (ISLs). Somewhat surprisingly, multi-target attacks that hamper connectivity between large regions are also feasible, at somewhat higher cost, but *without* increasing detectability compared to the single-target case.

In practice, LSNs may leverage multipath routing, potentially with randomized load balancing. We thus introduce a probabilistic version of ICARUS, that allows them to congest the target(s) with high probability, despite the uncertainty of routing. We analyze this attack against four intuitive routing schemes. Unfortunately, our experiments show that ICARUS is still able to attack targets at some additional cost compared to the single shortest-path scenario. The routing scheme that is most effective at increasing the attack cost—by 385% for the median target, *if* the adversary desires to minimize risk of detection—incurs a large latency increase on its paths, $1.32\times$ in the median and up to $2.04\times$ in more extreme cases. Despite this latency inflation, in the absolute, this increased attack cost still translates to less than 80 Gbps of attack traffic, and with a few tens of Mbps of bandwidth per bot, as promised by LSNs, translates to a few thousand bots. This is also the most pessimistic scenario for the attacker: there is no benign network traffic to lower the bar for congesting targets, the

LSN is willing to accept a large latency degradation, and the attacker seeks to minimize detectability rather than cost. In many settings, a few hundred bots will suffice instead.

Lastly, we briefly analyze additional attack opportunities that may arise from the dynamic nature of LSNs — as satellites move and paths change, there is natural flux in both the latencies of end-end paths, and in the traffic carried by each ISL. An adversary could potentially leverage latency changes to double the bandwidth of attack flows for a short period, in a manner similar to temporal lensing [55]. However, early results indicate little additional utility from this seemingly promising method. Further, an adversary may also time their attack flows to coincide with natural surges in link utilization from path changes. Our preliminary analysis indicates promise for this latter strategy, but a packet-level analysis is necessary to ascertain its impact.

Our main contributions are:

- We draw out the unique features of LSNs that introduce a vulnerability to DDoS, motivating a new analysis of a problem well-studied in terrestrial networks.
- We present the ICARUS attack, which exploits these unique features to successfully congest target links in an LSN. ICARUS addresses the new challenges the LSN setting poses for the adversary in terms of cost and detectability.
- We use the simple single shortest-path routing setting to develop intuition, and to show how connectivity between large regions may be hampered by ICARUS.
- We extend ICARUS to a randomized multipath routing scenario, and evaluate it against four intuitive routing schemes, showing that the attack still succeeds with high probability.

To the best of our knowledge, ICARUS is the first volumetric DDoS attack on upcoming LSNs. Beyond our particular attack methods and results, we hope that our analysis framework will aid further work on the vulnerability of LSNs to DDoS attacks, and on developing and evaluating mitigation strategies, e.g., routing that optimizes the latency-resilience tradeoff. Our framework is available on [GitHub](#).

2 Background & related work

Our contributions lie at the intersection of the nascent research on large upcoming LSNs, and well-studied (for terrestrial networks) denial-of-service (DoS) attacks. We introduce the relevant background for both.

2.1 LEO satellite networks

In the proposed and under-deployment LSNs, hundreds to thousands of satellites will be arranged in equally-spaced *orbital planes*, each containing the same number of satellites. Satellites connect to terminal units (TUs) on the ground with radio links. At the same time, satellites may connect to each other by laser ISLs. Each orbit has a fixed *height* above sea level; under 2000 km is considered LEO. This relative closeness to the ground provides a great advantage in terms of communication latency, as the signal has to travel a much

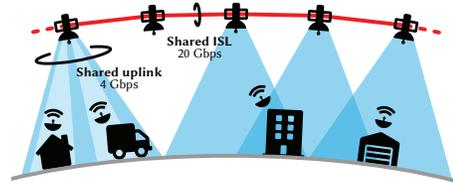


Figure 1: Satellite forwarding. Satellite-enabled hosts communicate with the satellites overhead using radio up- and downlinks. Traffic is routed from satellite to satellite via ISLs.

shorter distance to reach the satellite than, for example, to reach Geostationary satellites at 35,786 km.

In most proposed constellations, the orbital planes do not intersect the Equatorial plane at a 90° angle, and thus do not transit over the Earth’s poles. Instead, a lower *inclination angle* is used to allow the satellites to spend more time at lower latitudes, thus improving coverage above more populous areas, whilst avoiding the polar regions.

Our analysis throughout uses the first shell of the SpaceX Starlink constellation, for which the above parameters are: 72 orbital planes, each with 22 satellites, a height of 550 km, and an orbital inclination of 53° [60, 61, 63].

Satellite-ground connectivity. Figure 1 shows a typical LSN communication scenario. Terminal units (TUs) are installed at the clients on the ground. Each satellite can simultaneously connect with multiple TUs, thanks to their multi-beam antennas [17]. We call the aggregate capacity incoming to and outgoing from a satellite its *uplink* and *downlink* respectively. Based on recent FCC filings [60, 61], our analysis assumes both up- and downlink capacities to be 4 Gbps.

TUs can communicate with overhead satellites that are above a *minimum elevation angle* over the horizon. The angle varies from constellation to constellation, and expresses a trade-off between the size of the coverage area or *footprint* of a satellite, and the rate of communication achievable. Greater elevation angles translate to smaller footprints, but also force shorter paths inside the atmosphere and allow for better frequency reuse, thus increasing capacity. For Starlink’s shell I, the minimum elevation angle is planned to be 40°.

Inter-satellite connectivity. A key feature of the proposed constellations is the use of laser inter-satellite links (ISLs) on each satellite.¹ A typical approach is to have 4 ISLs per satellite, two connected to fore and aft satellites in the same orbital plane, and two to satellites in adjacent orbital planes [69]. The resulting ISL topology is used to transit traffic between TUs. As shown in Fig. 1, traffic can be forwarded from a TU to a satellite in view through an uplink, then across multiple ISLs, and finally to the destination via a downlink.

Some constellations, particularly during early stages of deployment, may forego ISLs, instead transiting data through a series of up and down transmissions between ground stations

¹SpaceX recently launched ISL-capable satellites [59], and Telesat has started producing them [65].

and satellites; this is, for instance, the case for Starlink in its first stages of deployment. However, ISLs are expected to be a key component of future LSNs, including more mature deployments of Starlink, and are crucial to achieving many of their benefits [10, 28, 29]. We therefore focus our analysis on the setting with ISLs; no-ISL LSNs are a strictly easier attack target: ground–satellite links will have a much lower capacity than ISLs, with each satellite supporting a few Gbps of uplink at most, simplifying the adversary’s goal of creating congestion. On the other hand, previous work [17] and laser equipment vendors’ offerings [47] suggest that ISLs will be able transfer up to 20 Gbps full-duplex, greatly increasing the capacity of the network.

Benefits: global low-latency broadband. With their hundreds to thousands of satellites, each providing multi-gigabit connectivity, LSNs can blanket the globe with broadband Internet. Besides their potential to extend the Internet’s coverage to under-served regions, LSNs are also generating excitement due to their potential for low-latency connectivity even in already well-connected areas. Prior work [10] estimates that LSNs could reduce latency for long-distance routes by more than $2\times$ compared to today’s Internet. Two factors contribute to this reduced latency: first, the speed of light in vacuum is roughly 50% higher than the speed of light in the glass medium of optical fiber. Second, a series of ISLs can often provide a more direct path between TUs than what fiber can achieve on the ground.

2.2 Denial of service

The term *denial of service* encompasses a large class of attacks. DoS attacks can target different resources at different layers in the network. We are interested in discovering and analyzing the threats to communication that are peculiar to LSNs, and that arise from the characteristics we have presented in the previous section. During a volumetric DDoS attack, a *botmaster* directs the traffic of thousands to millions of *bots* [3], i.e., compromised hosts, towards a target element in the network, overwhelming it. The *botnet* is usually composed of bots distributed across the globe; and targets can be servers, end-hosts, routers, etc. Traditional DDoS attacks send traffic to a target end-point. Newer attacks like Coremelt and Crossfire are designed to congest in-network elements instead—their approach to send traffic between bots or to initiate connections to a variety of public servers defies traditional methods to classify traffic into legitimate and unwanted.

Coremelt & Crossfire. In Coremelt [64], the adversary tries to congest a target inter-domain *link* by making use of legitimate-looking flows generated between the N bots of a botnet. Given the knowledge of the $\binom{N}{2}$ paths between the bots, the adversary can initiate flows that will cross the target link. With sufficient resources, the adversary can successfully congest any link in the network.

Crossfire [31] expands on this idea. With Crossfire, the adversary is able to hinder communication between entire

regions of a network, as bots simultaneously connect to strategically selected servers, thus creating overload on the links leading into the region under attack.

Both attacks have been notoriously hard to mitigate, since the adversary uses indistinguishable, legitimate traffic. In § 7 we argue that ICARUS poses an even larger threat, as the few mitigations available against Coremelt and Crossfire cannot be directly employed.

3 ICARUS adversary model

The ICARUS adversary controls a botnet of compromised hosts equipped with TUs that connect to the LSN. These bots are exploited by the adversary to send traffic over the LSN to congest links and disrupt communications between other TUs controlled by benign users.

3.1 The adversary’s objective

The adversary seeks to generate legitimate-looking traffic that, by careful selection of source and destination bots, disrupts target communications by exhausting the capacity of certain LSN links. More concretely, the adversary may target:

- *A satellite uplink or downlink.* This is the easiest attack, as these are the lowest bandwidth links in an LSN.
- *An ISL.* Congesting ISLs is a potentially more disruptive attack, as they carry traffic for more source-destination pairs. However, they are also more difficult to attack than uplinks and downlinks, as they have higher capacity.
- *Multiple links.* We also consider multi-link attacks, in which the adversary selects a combination of target ISLs and/or up/downlinks to achieve a broader attack effect. For example, the adversary could try to congest *all* the links in the load-balancing set for a target source-destination communication, or to target all links connecting two large terrestrial areas.

Our simulations label a link *congested* when aggregate traffic demand across it exceeds its capacity. In practice, degradation of communication starts before this threshold.

3.2 Metrics of attack success

For any of the above targets, the adversary is interested in causing disruption at *low cost* and while *avoiding detection*.

Cost. Attack cost measures the resources an adversary has to deploy to successfully achieve their attack’s goals. We express cost as the traffic volume (in Gbps) the adversary has to generate to be successful. With each bot generating tens of Mbps of attack traffic, as seen in initial measurements from Starlink [5, 11], an ISL’s 20 Gbps worth of capacity translates to, e.g., 500 bots each sending 40 Mbps.

Detectability and maxUp. If the LSN operator sees large changes in how much traffic a certain satellite receives on its uplink, the operator could potentially localize and identify the compromised bots. Thus, to reduce detectability, the adversary would distribute attack traffic across numerous uplinks. To characterize detectability, we use the *maximum absolute*

bandwidth increase caused by the attack traffic across the uplinks of the LSN (maxUp). We use the absolute increase, rather than relative, to obtain comparable results from simulations with different baseline traffic models.

3.3 The adversary’s capabilities & constraints

Unlike terrestrial networks, many aspects of an LSN’s operations are public knowledge or can be inferred.

LSN topology. Satellite orbits are public-domain information constantly updated and published by NORAD [12]. From these, an adversary can precisely compute the positions of the satellites for a chosen attack time; the error in such computations is at most a few kilometers per day into the future [32]. As noted earlier (§2), the ISL interconnect is expected to be a typical cross pattern. The design capacity of all links is also known through public regulatory filings. Thus, we assume full advance knowledge of the LSN topology. Moreover, changes in the shortest paths happen on timescales of seconds. As the maximum forwarding latency in the network is about 125 ms, the adversary can look at the LSN in snapshots [10], and compute the attacks for successions of snapshots. They can thus avoid the complexity of continuous satellite motion.

Routing. The topology is known, as is the propagation delay across each link at all times. Therefore, given frequent measurements of latency between a large set of bots, the adversary can determine how routes are being chosen. This is a much simpler setting for network tomography than one where the topology and forwarding latencies are unknown.

If routing is deterministic, e.g., shortest-path routing, the adversary can compute the full forwarding path for any source-destination pair ahead of time. If routing is randomized, e.g., for load balancing across near-shortest paths, we assume the adversary knows the algorithm with which the *load-balancing set* is constructed. The adversary therefore knows in advance the possible paths traffic will take, but not the actual path selected for forwarding a particular flow.

Bot locations & availability. A key promise of LSNs is global coverage, thus providing diverse locations for potential bots. Even if terminals themselves are secure, it suffices to compromise the user devices connected through them, and therefore exploits used today to create botnets directly apply in the case of a satellite-enabled botnet. We thus assume that the adversary can compromise TUs at any location on land. Given that such TUs, even today [66], feature GNSS²-based self-location to aid signal acquisition from satellites, the adversary accurately knows the locations of their bots.

Regarding the size of the botnet, we do not constrain the number of compromised TUs at any location beyond the limits imposed by the capacity of uplinks. We will later show that successful attacks only require hundreds to thousands of compromised TUs—a very small fraction of the tens of millions of TUs expected to be available globally (SpaceX has sought permits for 5 million TUs in the US alone [62]).

²Global Navigation Satellite System.

Attack control channel. We assume that the adversary can coordinate the bot army through a command-and-control channel, which is typical for modern botnets. As we show later, the advance knowledge of the topology, the routing algorithm, and the bot locations, allows the adversary to pre-compute and disseminate to the bots their attack traffic commands ahead of time. Fine-grained time-synchronization across bots would only be needed if the adversary sought to create extremely short-term congestion (e.g., for tens of milliseconds) [36, 55]. This would easily be possible through GNSS.

3.4 Unique attack opportunities & challenges

The above discussion highlights that an adversary targeting LSNs has several advantages compared to one targeting traditional terrestrial networks.

- An adversary has full access to the network topology. The positions of the satellites are published regularly [12], and can be predicted with high accuracy into the future [32]. The design detail of the interconnections, moreover, can be deduced from the FCC requests that the satellite operators are mandated to file. In contrast, the topology of terrestrial networks is concealed, and may even be obfuscated [44].
- Given the LSN’s global exposure, the adversary can recruit bots in diverse locations for generating attack traffic.
- The low-latency goal of LSNs leads to routing predictability, as shortest or near-shortest paths must be used.
- For already well-connected regions, the primary value of an LSN is low latency, as terrestrial fiber routes will be cheaper. This lowers an adversary’s burden: between a target source-destination pair, the adversary must only deteriorate a small set of desirable (low-latency) paths, instead of needing to congest a cut in the network graph.
- Our analysis of the topology and path structure of an LSN shows that certain links are more vulnerable than others, providing easy targets for an adversary.
- Exploiting the system’s predictability, communication with the bots can be asynchronous, making traditional detection of the command and control traffic difficult.

However, some aspects of the LSN setting also pose challenges to the adversary:

- Each bot has limited resources, and the pool of bots available is only a small fraction of Internet-connected hosts that could otherwise be purposed as bots. It is thus even more crucial for an adversary to keep attack cost low.
- The adversary needs to ensure that their limited attack resources are not wasted by self-congestion, whereby traffic from its bots congests links before reaching the target links.
- Congesting some satellite’s uplink exploiting a large number of bots directly underneath is counter to the adversary’s need to avoid detection. The LSN is a centrally-managed, intra-domain network, and therefore has better monitoring and policing capabilities than terrestrial transit networks.

Thus, the adversary has to be especially stealthy, and employ more distant and diffuse collections of bots.

- If the LSN uses randomized load balancing across multiple paths, this results in routing unpredictability for the adversary, requiring a probabilistic approach that would potentially consume more resources.

Our attack methods exploit the aforementioned unique opportunities, while addressing the new challenges.

4 ICARUS attack: shortest-path routing

We first discuss our ICARUS denial-of-service attack against an LSN in the simpler setting, where the LSN uses single shortest-path routing. While this provides low latency, the deterministic routing aids the adversary.

4.1 Attack mechanism

Paths in an LSN are typically stable on the order of seconds (§ 3.3). Thus, the adversary can compute the flows the bots must send on a static snapshot of the network, and use the same flow assignment for seconds before issuing a new one. Note that the adversary is also aware of the times when paths will change, and can plan their flow assignments for change-free periods. The assignments can be sent asynchronously to the bots for several periods in advance, as the entire system is fully predictable on a much longer timescale. For a particular system snapshot, the set of attack flows is computed as follows, similarly to Crossfire [31]:

- A1** *Link and path discovery.* The adversary uses the known inter-satellite topology to create a connectivity graph. They compute all the satellite in view of each bot, and add an uplink and a downlink for each reachable satellite to the connectivity graph. Finally, the adversary runs Dijkstra’s algorithm for all the $N(N - 1)/2$ pairs of bots to find the forwarding paths.
- A2** *Path filtering.* For a given target link or links, the adversary finds all bot-pairs whose shortest path traverses the target(s). The adversary retains only the paths that traverse the target(s) in the desired (attacked) direction.
- A3a** *Feasible attack flow computation.* The adversary must decide how much traffic to send on the chosen attack paths between its bot-pairs. The adversary checks if there exists a flow assignment such that (i) the target links are just above capacity, and (ii) no other links are congested. Satisfying these constraints ensures that the target(s) can be congested without the attack traffic self-congesting, i.e., being bottlenecked before reaching them. This approach minimizes cost in terms of attack bandwidth.
- A3b** *Reducing maxUp.* If **A3a** produces a feasible attack flow, the adversary iteratively optimizes maxUp. Recall that maxUp, D , is the maximum attack traffic volume across the uplinks. At each iteration i , the adversary lowers the permissible attack traffic volume on all uplinks, D_i ; then they perform the feasibility check again. This process

is repeated until the minimum value, D_{min} , is found, for which the attack is still feasible.

4.2 Evaluation setup

We evaluate the above attack strategy in terms of how its cost and maxUp depend on the attack scenario. The attack scenario varies the targets, i.e., either individual links or sets of links, and what type of benign traffic already uses the LSN.

Throughout, we use simulations on the SpaceX Starlink shell I constellation (§2) with a total of 1584 satellites. Note however, that our evaluation framework supports arbitrary constellations; an example set of results for a different configuration is included in §D in the Supplemental materials. We assume that bots and benign TUs can be located at any point on land. To keep the simulations tractable, we discretize the continuous space of geolocations to a regular geodesic grid. This is done in the form of a triangular tiling of the planet that, when restricted to land areas, amounts to a total of 1761 possible positions. (The restriction to land areas is driven by our below models of benign traffic, which are based on population and economic activity.) Each of the triangular tiles represents an area of 100,000 km². Since a single Starlink satellite covers an area ten times this size, this grid suffices to capture the nuances of different TU placement relative to the same satellite.

We test three traffic scenarios for benign traffic on the LSN:

- *Empty network:* This case, with no benign traffic, serves as a baseline, comparing against which, we can understand the impact of existing traffic.
- *GDP traffic:* A “gravity” sampling model, where probability of adding 10 Mbps of benign traffic between two locations is proportional to the product of their Gross Domestic Product (GDP) [48]. Each tile’s centroid serves as a traffic source/sink location, with the aggregate GDP for the tile’s area being this location’s GDP weight. We then sample 250k times, discarding the samples that, if added, would exceed 90% of the capacity on the links. This process results in a utilization of 28% of the on-land uplink capacity. This model treats traffic as proportional to economic activity, representing the view that the biggest economic centers will drive traffic and revenue.
- *Pop traffic:* Another gravity model similar to the above, except the population is used as the weight instead of GDP [13]. This model represents the scenario where the LSN’s primary role is improving coverage in regions that are populous but still poorly connected. For this model, we get 34% utilization.

All the simulations in the following are executed by our [LSN simulation framework](#) (~5000 lines of Python code).

4.3 Single link target scenarios

Attacking an uplink. The outcome of the attack on an uplink is easy to determine: either the adversary has enough bots in the area of coverage of the victim satellite uplink, or

the attack cannot succeed. This attack has therefore maximum maxUp in our model, as the adversary has to completely fill the uplink with attack traffic.

Downlinks. Unlike uplinks, downlinks can be reached from arbitrarily afar, and therefore the adversary can more easily find bots to congest them. In theory, the attack flows can self-congest: as the attack traffic flows towards the target downlink, more and more flows join paths, filling the ISLs leading up to the target. While this can violate constraint ii) in A3a (§ 4.1), it is unlikely for an attack on a downlink — all the ISLs preceding the target downlink can carry $5\times$ the bandwidth needed to congest the downlink. As a result, our simulations reveal that *all* downlinks from satellites located over land are attackable in this manner. As our model does not allow TUs in the seas, there is no way to congest downlinks for the remaining satellites. We also verified that this result remains the same regardless of our three traffic models, as even with benign traffic, there is always enough leftover capacity in the network for the relatively small amount of attack traffic bandwidth required.

ISLs. ISLs have higher bandwidth than up-/down-links, making an attack more complicated and more costly to achieve. Not only is more attack traffic needed to cause congestion on an ISL compared to a downlink, but also self-congestion *does* often occur in practice for attack flows headed to a target ISL. This often results in there being no feasible attack flow. Consequently, of the 6336 possible ISL target links (1584 satellites, each with 4 ISLs, counting both directions), we find that in an empty network scenario, 5470 (86%) are feasible. With GDP or Pop traffic, this drops to 72% and 71% respectively. This is along expected lines: benign traffic reduces the available capacity on the attack flow paths, limiting the attack traffic that reaches the target.

An interesting finding from our simulations is that with all three models, of the end-to-end paths with at least one ISL, fewer than 0.6% do not contain at least one ISL the adversary can congest. Therefore, this attack can disrupt >99.4% of all communications that rely on an ISL.

Cost and maxUp. In the above, we have only discussed feasibility for each individual targeted link: for uplinks, feasibility depends simply on having enough bots under the target uplink; all downlinks over land are feasible to attack; and for ISLs, self-congestion limits feasibility, with the details dependent on the benign traffic. For the feasible attacks, we can also analyze attack cost and maxUp.

Figure 2 shows the cost and maxUp of attacks against all links that are feasible to attack (downlinks and ISLs together; uplinks are uninteresting as discussed, and thus omitted) for all three traffic models. From the cost graphic (top), we see that for the empty network, there are only two modes in the CDF — all downlinks cost 0.2 and all ISLs cost 1 to attack. (We present costs normalized to one ISL’s bandwidth, i.e., 20 Gbps; recall that downlinks have one-fifth the capacity of an ISL.) This is as expected: in an otherwise empty network,

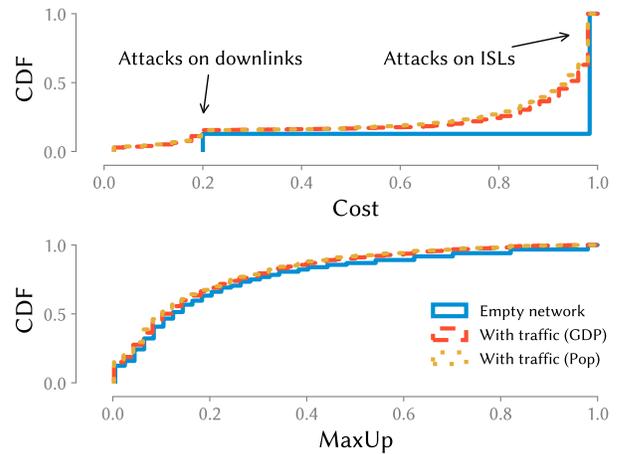


Figure 2: Cost and maxUp of single-target attacks. While maxUp is similarly distributed either with or without baseline traffic, attack cost decreases significantly.

congesting any link requires sending the target link’s capacity worth of traffic. In the presence of benign traffic, for both traffic models, the attack cost is lower, as the adversary needs to add a smaller amount of traffic to the existing benign traffic. In terms of number of bots, a cost of 1 translates to a few hundred bots sending tens of Mbps each (§ 3.2). The maxUp plot (Fig. 2, below) shows the distribution of maxUp across attacks on different target links. We normalize maxUp to one uplink’s bandwidth, such that needing to fully saturate a satellite’s uplink is the worst case for the adversary, and means that maxUp is 1. The maxUp of an attack on the median target link is below 0.13 in the empty network scenario. Thus, the *maximum* satellite uplink bandwidth consumed across attack flows sent towards these target links comprises roughly one-eighth of an uplink’s bandwidth. Adding benign traffic to the network with either traffic model further lowers the adversary’s risk of detection.

While adding benign traffic decreases attack cost and lowers maxUp for those target links that are feasible to attack, it actually decreases the fraction of feasible target links. However, closer inspection shows that this is not a particularly severe problem for the adversary: the links that become less vulnerable when benign traffic is present are mostly above the oceans, and not the higher-value ones over the more populous regions (further analysis is presented in §A.1 in the Supplemental materials).

4.4 Multi-link region disconnection scenarios

Beyond attacking individual links and communications that traverse them, an adversary may seek to hamper all communication between two large geographic areas. To do so, they must congest *all* paths connecting the two regions simultaneously. This makes the attack more challenging: the adversary now has to carefully select the set of target links to congest, such that hampering region-region connectivity incurs low cost and has low maxUp. Figure 3 shows an example in which

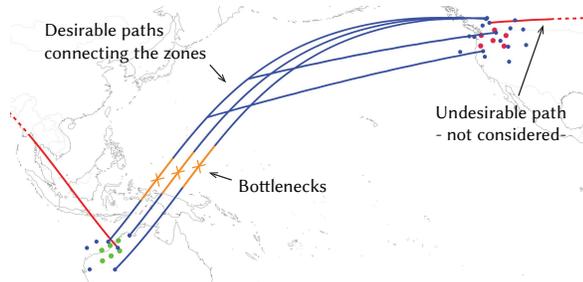


Figure 3: Example of region disconnection. Although 12 distinct shortest paths are available between North-western America and North-western Australia, just 3 bottlenecks suffice to prevent communication between these large zones.

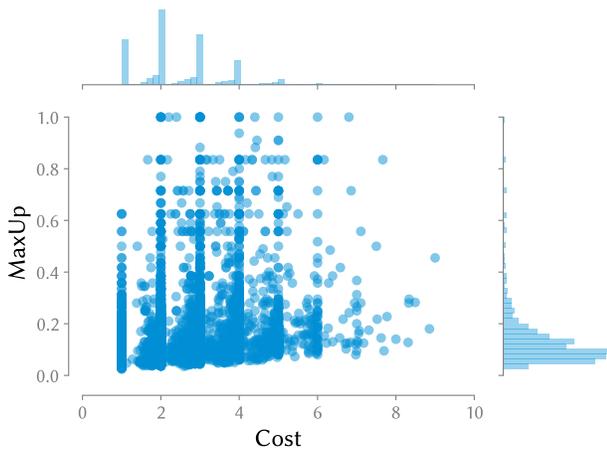


Figure 4: Cost and maxUp of zone disconnection attacks. As shown by the marginals, maxUp is low for most attacks. The peaks shown in the cost marginal correspond to the number of bottlenecks in the paths between the zones.

the adversary intelligently selects 3 bottlenecks shared across the 12 different paths, greatly reducing the attack cost. In many cases, however, the choice is not as obvious.

The adversary has to select the minimum set of links *it is able to attack simultaneously*, among the ones in the paths connecting the zones, such that their congestion results in a complete disconnection. Unfortunately, this problem is equivalent to the minimum set cover problem (we prove equivalence in §B in the Supplemental materials), which is NP-hard [34]. To avoid exponential explosion, we use a heuristic which computes a good approximation of the set of links to attack in polynomial time. Each link in the paths to congest is assigned a score, equal to the ratio between the maxUp of the attack on the link, and the number of paths that share the link. The lower this score, the stealthier it is to attack such a link. We therefore add the link with the lowest score to the attack set, we remove it from the network alongside all the paths it congests, and finally recompute the scores for all remaining links. We continue this procedure until all paths are congested. We run this heuristic three times, slightly varying the order in which links are removed, and keep the lowest maxUp result.

Simulation results. We iteratively sample 5000 pairs of regions, each comprising six points on the geodesic grid (§ 4.2). For each pair, we run the attack twice, inverting source and destination region, for a total of 9658 viable attack cases (some are discarded because the zones are overlapping). The results are shown in Fig. 4.

We find that 9208 of these zone pairs (95%) can be successfully attacked. The median maxUp of these attacks is 0.10, implying that attacking large regions does not expose the adversary more than the single-link attacks (where the maxUp was 0.13). The minor difference in maxUp arises because the zone construction algorithm prevents some of the corner cases that increase the median maxUp in the single-link attacks. Since zones are composed by multiple grid points, they have to be located farther away from the edges of satellite connectivity (high latitudes, or narrow corners of continents), thus giving the adversary more uplinks on which to distribute the attack traffic. Finally, the overall cost is mainly driven by the number of bottlenecks found (characterized by the spikes in the marginal in Fig. 4), which is consistently below 4.

5 ICARUS against more sophisticated routing

We use the single shortest path routing setting discussed thus far primarily to develop intuition about the vulnerabilities of an LSN, and to broadly understand an adversary’s tradeoffs. In practice, the more likely scenario entails the LSN load-balancing traffic across multiple paths. We therefore consider several candidate routing schemes for such load balancing, and then formulate and evaluate a modified attack strategy suited for this more complex setting.

5.1 Candidate routing schemes

Developing, evaluating, and comparing new routing strategies is not our objective, and has been tackled in some depth in prior work (§ 8). However, to understand how load balancing across multiple paths impacts an adversary, we must frame some suitable routing methods. Here, we use one guiding constraint: the chosen non-shortest paths must still be “near shortest”, i.e., not increase latency by a large amount. In the extreme, any path’s latency must not exceed that of a terrestrial fiber route.³ We thus evaluate the following multi-path routing variants:

- k*-SP** *Shortest paths:* This strategy load-balances traffic among the *k*-shortest source–destination paths.
- k*-DG** *Ground-to-ground disjoint paths:* This strategy only considers the *k* shortest paths that are node-disjoint, i.e., without shared uplinks, downlinks, or ISLs.
- k*-DS** *Satellite-to-satellite disjoint paths:* This strategy is similar to the above, with the exception that overlap is allowed on uplinks and downlinks. Disjointness is enforced exclusively on the ISLs.

³This fiber latency is estimated, based on past measurement work [58], by multiplying the great-circle distance by a path-stretch factor of 1.53, and then dividing by the speed of light in fiber, $\approx 2c/3$.

***k*-LO** *Limited-overlap shortest paths*: we implement the ESX algorithm by Chondrogiannis et al. [15], a heuristic that finds the shortest k paths with a *similarity score* of no more than 50%.

Note that sometimes these algorithms yield fewer than k paths because of the faster-than-fiber and path disjointness constraints. (For k -SP, only the former applies, and is only limiting for nearby end-points.) Each algorithm considers different types of near-shortest paths, varying the degree to which paths overlap. k -SP, which does not limit overlap, typically offers multiple paths with nearly the same latency as the shortest, but risks the shared links across these paths becoming the bottleneck. On the other extreme, the most restrictive scheme, k -DG, typically does not even provide more than 4 paths. Further analysis of the path structure of these algorithms is in §C in the Supplement. For each scheme, we assume that the LSN uses randomized load balancing (similar to ECMP) across the chosen path. Note that uniform random load balancing is the worst case for the adversary; any deterministic adaptive scheme will strictly reduce uncertainty for the adversary.

5.2 Probabilistic ICARUS

The use of load-balanced, uniformly randomized routing introduces uncertainty in the *link discovery* phase. As the network chooses the end-to-end path only at forwarding time, from a set of k pre-computed paths, the adversary cannot know in advance which specific path will be taken by their attack traffic. We therefore present a probabilistic variation of ICARUS that accounts for this uncertainty. This attack performs attack-flow assignment in such a way that the target link will be flooded with high probability, while avoiding self-congestion among attack flows, and lowering maxUp.

Congesting the target. The adversary pre-computes all the load-balancing paths for all its bot source-destination pairs, and considers those (s, d) -pairs for which *at least* one path contains the target link. Consider an (s, d) -pair connected by $n \leq k$ paths, with m of them crossing the target link. Then the event “the forwarding path chosen for this (s, d) -pair uses the target link” can be described as a Bernoulli random variable $X_{(s,d)}$ with probability of success $p_{(s,d)} = m/n$. Suppose the adversary uses an attack-flow set \mathbb{A} , comprising many (s, d) -pairs. Then the sum $Y = \sum_{s,d \in \mathbb{A}} X_{s,d}$, is a random variable describing the number of attack flows that are forwarded across the target link. The X variables are independent but *not* identically distributed, with different m and n across (s, d) -pairs, therefore Y follows a Poisson binomial distribution.

The adversary’s goal is for the attack-flow set, \mathbb{A} , to be such that at least T attack flows use the target link with high probability. Here, T is determined by each attack flow’s bandwidth, and the target link’s capacity; in our model, for example, an ISL has a capacity of 20 Gbps; if a single (s, d) -pair can transmit 40 Mbps (3.2), T is 500.

The target link is congested if $Y \geq T$. The adversary wants to bound their probability of failure to at most a small

value, β , i.e., $P[Y < T] \leq \beta$. The adversary can get more certainty, i.e., lower β , at the expense of more resources.

The computation of $P[Y < T]$ is infeasible for non-trivial settings, but we can approximate it via the Chernoff (lower tail) bound [46]:

$$P[Y < (1 - \delta)\mu] \leq \exp(-\mu\delta^2/2) \quad (1)$$

where $\mu = E[Y]$. By setting $\delta = 1 - \frac{T}{\mu}$ in eq. (1), we get:

$$P[Y < T] \leq \exp(-\mu(1 - T/\mu)^2/2)$$

Note that $\mu = E[Y] = \sum_{(s,d) \in \mathbb{A}} p_{(s,d)}$ by the linearity of expectation. The adversary can easily calculate μ for any attack-flow set, \mathbb{A} . To ensure that $P[Y < T] \leq \beta$, the adversary can pick \mathbb{A} ’s constituent (s, d) -pairs such that they satisfy:

$$\exp(-\mu(1 - T/\mu)^2/2) \leq \beta$$

For $0 < \beta < 1$ and $1 \leq T < \mu$, this is equivalent to:

$$\mu > \sqrt{\ln \beta \cdot (\ln \beta - 2T)} + T - \ln \beta = \mu_{min} \quad (2)$$

Avoiding self-congestion. The adversary must also ensure that the attack flows do not cause self-congestion, i.e., congest other links before reaching the target. This problem, of *not* congesting non-target links, requires the same machinery as that above for congesting the target. For every non-target link, l , we simply need to repeat analysis similar to the above, except towards ensuring we *do not* congest l , i.e., $W_l < C_l$, where W_l is a random variable describing the number of attack flows that traverse l , and C_l is l ’s capacity.

Say γ is a parameter analogous to β above, indicating the maximum acceptable risk of self-congestion. For any non-target link, l , μ_l (analogous to μ above) is the expected number of attack flows crossing l , such that $\mu_l = \sum_{(s,d)} p_{(s,d)}$ for (s, d) -pairs that traverse link l . Analysis similar to the above—but considering the upper tail—yields, the following condition:

$$\mu_l < \frac{1}{2} \left(-\sqrt{\ln \gamma \cdot (\ln \gamma - 8C_l)} - \ln \gamma + 2C_l \right) = \mu_{l,max} \quad (3)$$

Probability of attack success. An attack is successful if the target is congested without congesting any other link. We combine the probabilities computed so far as follows:

$$P[\text{success}] = 1 - P[\text{failure}] \quad (4)$$

$$= 1 - P[Y < T \vee W_1 \geq C_1 \vee \dots \vee W_N \geq C_N] \quad (5)$$

$$\stackrel{\text{u.B.}}{\geq} 1 - \left(P[Y < T] + \sum_{l=1}^N P[W_l \geq C_l] \right) \quad (6)$$

$$\geq 1 - (\beta + N \cdot \gamma) \quad (7)$$

$$= 1 - \alpha \quad (8)$$

where eq. (6) is given by the union bound, and N is the number of non-target links. For a high probability of success, α has to be small. In our setting, N is in the range 1500–2000. We therefore choose $\gamma = 1/N^2 = 1/2000^2$ and $\beta = 0.1$, which gives $P[\text{success}] \approx 0.9$. Empirically, we find that the Chernoff bound is quite loose, and for these values of β and γ the success probability is higher ($\geq 95\%$).

Attack set construction. The adversary must construct the attack-flow set, \mathbb{A} , such that it simultaneously satisfies the constraints on μ , and for every non-target link l , μ_l :

$$\mu = \sum_{(s,d) \in \mathbb{A}} p_{(s,d)} > \mu_{min} \quad \text{and} \quad (9a)$$

$$\mu_l = \sum_{(s,d) \in W_l} p_{(s,d)} < \mu_{l,max} \quad \forall \text{ non-target links } l \quad (9b)$$

Since μ is additive, we use a greedy approach:

- We sort the (s,d) -pairs by decreasing value of $p_{(s,d)}$.
- Iteratively, we add an (s,d) -pair to \mathbb{A} , checking that eq. (9b) holds. Otherwise, we discard the pair and move on to the next one.
- We repeat this until either eq. (9a) is satisfied—in which case the attack succeeds with probability $\geq 1 - \alpha$, or there are no more (s,d) -pairs, and the algorithm fails to guarantee this probability of success.

The above approach yields the attack set with the minimum number of (s,d) pairs. However, maxUp is not minimized, and is (nearly) μ_{max}/C_{uplink} . If the adversary wants to minimize maxUp before cost, they can iteratively lower $\mu_{l,max}$ for the uplinks to the minimum value such that the construction of \mathbb{A} still succeeds.

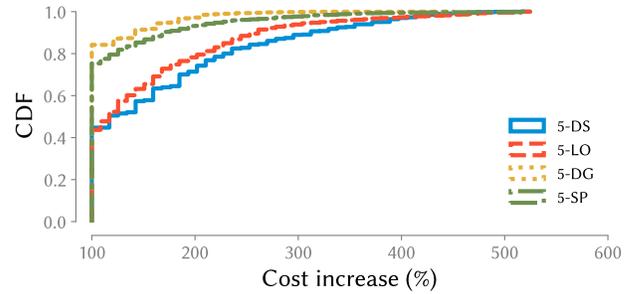
Further optimization. Due to the structure of the topology and the routing schemes, for each target link, there are certain (s,d) -pairs for which $p_{(s,d)} = 1$. In the above probabilistic analysis, such (s,d) -pairs needlessly contribute to making the probability bounds loose. Instead, they can be considered separately, and the above analysis done after accounting for those (s,d) -pairs.

5.3 Evaluation

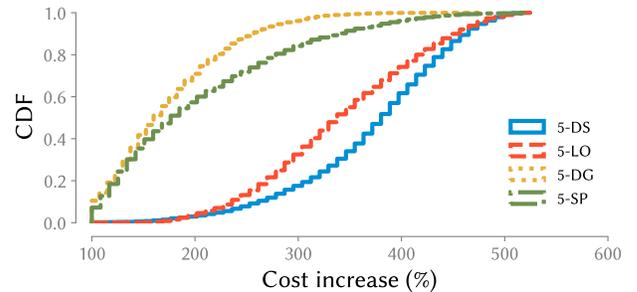
We evaluate probabilistic ICARUS using the same constellation, Starlink shell I, under the 4 routing schemes from § 5.1, with $k = 5$ paths, and uniform randomization across the paths. We present results for the empty network scenario, without any benign traffic, as it is the costliest for the adversary.

We target a maximum attack failure probability α of 10%, and measure the cost for the adversary in terms of the size of the attack set. A resilient routing scheme will force the adversary to incur higher attack cost for the same α , while vulnerable routing schemes will incur cost closer to the single shortest-path routing case. Following this intuition, we present attack costs for different routing schemes in comparison to the latter.

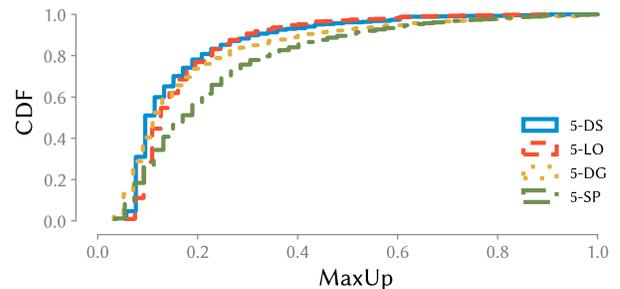
Fig. 5a and 5b show the attack cost for all 4 routing schemes, when the attacker optimizes for cost and for maxUp, respectively. Surprisingly, 5-DG and 5-SP perform similarly, even though they are very different: the former does not allow any overlap between paths for the same (s,d) -pair, while in the latter, path overlaps are not restricted at all. The commonality, however, is that the uncertainty in the forwarding path is low for both. In 5-SP, this is due to the lack of separation between the 5 absolute-shortest paths in the SN. In



(a) Cost CDF for the attacks, when executed with cost minimization. For maxUp, please refer to text in § 5.2.



(b) Cost CDF for the attacks, when executed with maxUp minimization. The cost increases significantly compared to the previous figure, to the benefit of maxUp (below).



(c) MaxUp CDF for the attacks, when executed for maxUp minimization.

Figure 5: Cost and maxUp of the attacks on all ISLs assuming different load-balancing strategies.

5-DG, on the other hand, the low uncertainty comes from the lack of alternatives: between many (s,d) -pairs, this scheme is overly restrictive and does not provide 5 paths. The adversary can use such (s,d) -pairs in the attack-flow set, effectively side-stepping routing uncertainty.

The most resilient scheme in terms of increasing the adversary's cost, is 5-DS. This scheme strikes a balance between avoiding path overlap and being flexible enough to maintain enough path options: enforcing disjointness between satellites is less limiting than ground-to-ground disjointness. The median cost increase compared to single shortest-path routing is 385% with 5-DS. The price of this resilience is increased latency — paths often incur more than $1.5 \times$ latency compared to the shortest path (Fig. 4 in the Supplemental materials).

Unfortunately, despite incurring a high latency cost, while

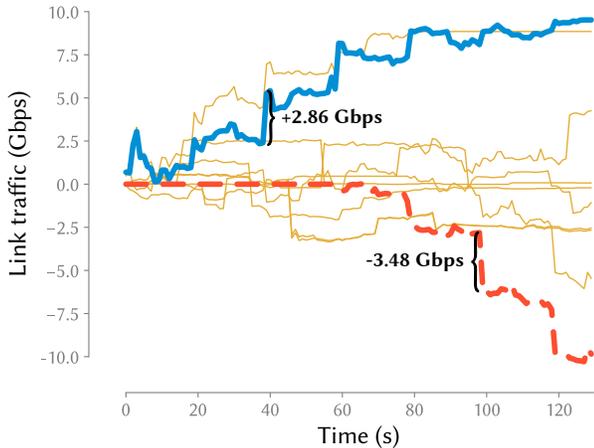


Figure 6: Load surges. The figure shows the link traffic over time of the 10 links with the largest load surges. The biggest positive (—) and negative (--) surges are also highlighted.

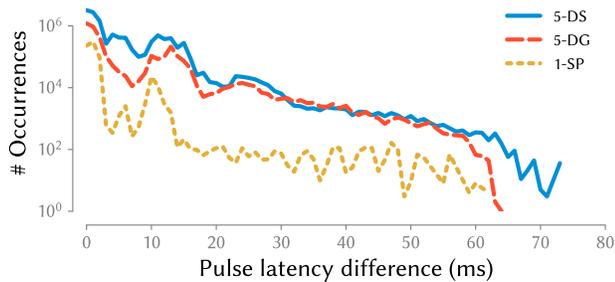


Figure 7: Feasibility of pulsing attacks.

the percent-increase in cost seems large (385%), in the absolute, this is still under 80 Gbps of attack traffic, and botnets with a few thousand bots (depending on per bot bandwidth) can still congest the median LSN link. Note also that this cost is in the most difficult setting, i.e., without any benign traffic, with the LSN accepting a large latency deterioration, and with the attacker seeking to absolutely minimize maxUp. Further, valuable links above land are easier to attack. Figure 2 in the Supplemental materials shows *where* the cost of the attack is highest across the network’s ISLs, with 5-DS. Attacking ISLs that are above land is easier because the adversary can always find a source-destination bot-pair for which there are few, or greatly overlapping paths (reducing uncertainty).

For maxUp, shown in Fig. 5c, 5-SP is the outlier, with higher maxUp than the other three schemes, for which results are similar. This is because other algorithms often have multiple uplinks available in the same load-balancing set, thus spreading the attack traffic and reducing its maxUp. In 5-SP, paths mostly overlap, and especially on the uplinks, preventing this effect.

6 Open problem: exploiting LSN dynamics

Thus far, the attacks we have presented consider only snapshots of the SN’s state, with the adversary drawing on the relative stability of the network on the timescale of seconds to

pre-compute attack flows. It is worth considering how/if the adversary may exploit the LSN’s dynamics as an additional opportunity, rather than a minor hindrance that requires per-computed, albeit frequent, changes in the attack-flow set. We discuss two potential opportunities of this type.

Load surges. The motion of satellites naturally moves ISLs across low- and high-utilization areas. As a result, the network’s links see large fluctuations in their utilization over time. Figure 6 shows how the load changes in time across the links with the biggest surges and drops over 130 seconds of simulation for the GDP traffic model. An adversary that knows or can reasonably guess the traffic matrix of the LSN can use these load surges to increase the effectiveness of low-volume attacks: well-timed bursts that coincide with the natural surges can cause high congestion, at a fraction of the cost. The effectiveness of such an attack depends on the the LSN’s protocol stack, e.g., how long does congestion control take to detect such short, transient congestion, and how does it react and recover? Addressing this requires packet-level analysis, and is left to future work.

Pulsing attacks. Another dynamic feature of an LSN is the time-varying latency of paths. Consider a bot that has a path to a destination, d , across a target link. Over time, this bot’s path to d changes, with some of these changes still traversing the target, but potentially over a higher- or lower-latency path, resulting from satellite motion. When the path changes from a higher latency one to a lower latency one, the adversary can potentially double the effective attack bandwidth of the bot: traffic sent on the first and the second path will traverse the target link *simultaneously* for a short time interval. This “pulsing attack” bears similarities to *temporal lensing* [55], with the important difference that temporal lensing uses *external* infrastructure like DNS to create a time difference between the forwarding paths, while in LSNs this is provided by the network’s inherent dynamics.

We briefly test the feasibility of pulsing attacks by running the following experiment:

1. At each time-step, we consider a target link, and compute all the paths across it.
2. We identify paths that will not be valid in the next time-step, and compute their replacement paths.
3. For each changed path, we compute the difference in latency from the source to the target link, between the original path and its replacement.

We run this procedure for all 6336 target links in a 130-sec simulation, and a time-step of 1 sec for evaluating changes. Figure 7 shows the results for a few routing schemes. For single shortest-path routing, achieving latency differences of tens of milliseconds across the path change is possible less than once per simulation second across the *entire* constellation. The multipath schemes increase this opportunity, but it is nevertheless small: even for the most favorable algorithm, 5-DS, there are only 8133 pulses with a duration above

50 ms. Since there are 6336 target ISLs, the adversary can benefit from little more than 1 pulse per ISL, on average, over the entire 130 sec of the simulation. Thus, while pulsing is possible in principle, its utility for an adversary is limited to occasionally impacting a small number of targets.

Future outlook. Our above analysis indicates positive potential for load surges, and a largely negative result for pulsing. Exploring other methods for an adversary to leverage LSN dynamism, as well as a deeper understanding of the impact of load surges, are left to future work.

7 Mitigations

We find that several traditional methods of addressing DDoS attacks are not applicable against ICARUS:

- *Overprovisioning* to absorb more than expected traffic is difficult: our model already accounts for ISLs having higher capacity than access links, and increasing ISL capacity or number will push against the cost, weight and power constraints of the satellites.
- *In-network filtering* [4, 26, 39, 42, 45, 54] requires computational resources, which are lacking at both the satellites and the TUs (TUs often need to be portable). More fundamentally, like Coremelt and Crossfire, attack traffic is indistinguishable from benign traffic. Therefore, even if satellites or TUs have the computational capabilities for filtering, it is unclear how they would distinguish malicious and benign traffic.
- *Capabilities* in the form of cryptographic tokens that provide access rights [2, 37, 38, 40, 52, 72] are not useful here either — the adversary compromises legitimate satellite-connected users to gain access to the LSN; thus inheriting these users' capabilities.
- *Cloud-based mitigations* [1, 16, 21, 24, 41, 51] offload the filtering of adversarial traffic to the cloud. Since TUs will often use the LSN as their last-mile provider, the burden of passing traffic through a cloud provider falls on the LSN, incurring additional bandwidth and latency overheads.

Thus, effective defenses against Coremelt and Crossfire—such as upgrading capacity or re-routing traffic to more capable networks—cannot be used to alleviate ICARUS' impact. In light of the above, we discuss two avenues that show promise towards neutralizing ICARUS.

Resilient routing and network design. Our experiments in § 5.2 show that resilient routing can greatly increase the attack's cost. While for our tested routing schemes, this comes at the expense of increased latency, we have only scratched the surface in exploring resilient routing; identifying routing schemes with the most favorable tradeoff curve remains an interesting direction for future work. Another possibility is to attack the problem using network topology design — Bhattacharjee et al. [10] have recently proposed to re-arrange ISLs, deviating from the traditional “cross” pattern, to improve forwarding latency. A similar strategy can be used in an

adversarial setting: altering the structure of the network can improve resilience to attacks with minimal impact on latency.

Traffic separation and differential pricing. In this work, we assume that all TUs are equally capable, and optimized for low-latency communication. It is foreseeable, however, that not all hosts will require such an optimization. This consideration opens up the opportunity of creating different classes of service, with a premium service offering the lowest latency and highest resilience, at the expense of shifting other traffic to longer, less predictable paths. It remains to be seen, however, whether this introduces enough randomness in forwarding to make the attacks too costly to perform, or if, instead, the higher resource usage caused by more circuitous paths *increases* vulnerability.

8 Other related work

We discuss related work not already covered in § 2 or § 7.

Attacks on satellite systems. The most commonly recognized threat to satellite communications on the physical layer is *jamming* [30, 49, 56]: the adversary uses high-power antennas to induce noise on up- and downlinks, preventing the endpoints from reaching the ingress satellite. ICARUS is however beyond detection by a jamming analyzer as it uses protocol-compliant traffic from authorized (albeit compromised) sources. GNSS spoofing is another well-understood threat, with a long list of attacks and mitigations [57]. More recently, the attention has shifted towards the security of cryptographic standards used in satellite communications [18], and lack thereof [50].

Network-layer denial of service on LSNs has been addressed, to the best of our knowledge, only recently by Usman et al. [67]. However, their focus is on traditional geostationary orbit satellite networks, using only bent-pipe connectivity through satellites, without any notion of ISLs. The authors thus focus on ping flooding to exhaust control plane resources at GS network devices. In contrast, our work examines a completely different setting with modern LSNs, analyzing an entirely different breed of attacks: congestion-causing volumetric DDoS attacks from distributed bots. To the best of our knowledge, we are the first to frame and study this problem.

Routing in satellite constellations. Given the dynamic nature of links in LSNs, many works focus on the challenges of discovering and disseminating network status information, and optimizing forwarding latency [14, 19, 20, 22, 23, 27, 43, 53, 68–71, 73]. Barrit et al. [6–8] bring software-defined networking (SDN) to LSNs, and introduce the Temporospatial-SDN. The SDN controller uses the predictability of satellite orbits to maintain an accurate view of the LSN's topology, and aid routing decisions. Despite this past work, the subject of optimizing routing for resilience against an ICARUS-like attacker remains an open question.

9 Conclusion

We present ICARUS, the first volumetric DDoS attack against next-generation LSNs. We demonstrate ICARUS's disruptive potential across a variety of scenarios, with different LSN routing schemes, and different attack targets. ICARUS's potency stems from leveraging several unique characteristics of LSNs, whereby the adversary operates with greater information than available for terrestrial networks, and exploits the topology and path structure of LSNs. ICARUS also turns the low-latency and global coverage objectives of LSNs into vulnerabilities. Nevertheless, successful attacks of such networks require careful limitation of cost and detectability (characterized with the maxUp metric), as ICARUS does. In addition, randomized multipath routing increases an adversary's cost, but as we experimentally show, ICARUS also succeeds in that setting. We hope that our first steps in understanding the vulnerability of LSNs to DDoS will seed the ground for further research on this topic. To this end, we release our simulation framework, which is the first tool for analyzing DDoS attacks and defences on LSNs. It allows easy testing of the resilience of arbitrary constellation configurations and novel path-selection algorithms.

Acknowledgements

We are grateful to our anonymous reviewers and our shepherd Gerd Zellweger for their insightful feedback and valuable suggestions. We gratefully acknowledge financial support from ETH Zurich and from the Zurich Information Security and Privacy Center (ZISC).

References

- [1] Amazon. Amazon AWS shield. <https://aws.amazon.com/shield/>, 2020.
- [2] Tom Anderson, Timothy Roscoe, and David Wetherall. Preventing Internet denial-of-service with capabilities. *ACM SIGCOMM Computer Communication Review*, 34(1):39–44, 2004.
- [3] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the Mirai Botnet. In *USENIX Security Symposium*, pages 1093–1110, 2017.
- [4] K. Argyraki and D.R. Cheriton. Active internet traffic filtering: Real-time response to denial-of-service attacks. *USENIX Annual Technical Conference*, pages 135–148, 2005.
- [5] Ars Technica. SpaceX Starlink speeds revealed as beta users get downloads of 11 to 60mbps. <https://arstechnica.com/information-technology/2020/08/spacex-starlink-beta-tests-show-speeds-up-to-60mbps-latency-as-low-as-31ms/>, 2020.
- [6] Brian Barritt and Wesley Eddy. Temporospatial SDN for aerospace communications. In *AIAA SPACE Conference and Exposition*, 2015.
- [7] Brian Barritt, Tatiana Kichkaylo, Ketan Mandke, Adam Zalcman, and Victor Lin. Operating a UAV mesh & internet backhaul network using temporospatial SDN. In *IEEE Aerospace Conference*, 2017.
- [8] Brian J. Barritt and Wesley Eddy. SDN enhancements for LEO satellite networks. In *AIAA International Communications Satellite Systems Conference*, 2016.
- [9] Debopam Bhattacharjee, Waqar Aqeel, Ilker Nadi Bozkurt, Anthony Aguirre, Balakrishnan Chandrasekaran, P. Brighten Godfrey, Gregory Laughlin, Bruce Maggs, and Ankit Singla. Gearing up for the 21st century space race. In *ACM Workshop on Hot Topics in Networks - HotNets*, 2018.
- [10] Debopam Bhattacharjee and Ankit Singla. Network topology design at 27,000 km/hour. In *International Conference on Emerging Networking Experiments And Technologies - CoNEXT*, 2019.
- [11] Business Insider. Starlink's speed tests may look impressive, but experts say SpaceX's satellite-internet project is unlikely to win any federal subsidies. <https://www.businessinsider.com/spacex-starlink-beta-speedtest-results-bandwidth-ping-latency-fcc-rdof-2020-8>, 2020.
- [12] CelesTrak. NORAD two-line element sets current data. <https://celestrak.com/NORAD/elements/>, 2020.
- [13] Center For International Earth Science Information Network-CIESIN-Columbia University. Gridded Population of the World, Version 4 (GPWv4): Population Count, Revision 11. <https://sedac.ciesin.columbia.edu/data/set/gpw-v4-population-count-rev11>, 2018.
- [14] Chao Chen, Eylem Ekici, and Ian F. Akyildiz. Satellite grouping and routing protocol for LEO/MEO satellite IP networks. In *ACM international workshop on Wireless mobile multimedia*, 2002.
- [15] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, Ulf Leser, and David B. Blumenthal. Finding k-shortest paths with limited overlap. *The VLDB Journal*, 29(5):1023–1047, 2020.
- [16] CloudFlare. Comprehensive DDoS protection. <https://www.cloudflare.com/ddos/>, 2020.

- [17] Inigo del Portillo, Bruce Cameron, and Edward Crawley. Ground segment architectures for large LEO constellations with feeder links in EHF-bands. In *IEEE Aerospace Conference*, 2018.
- [18] Benedikt Driessen, Ralf Hund, Carsten Willems, Christof Paar, and Thorsten Holz. Don't trust satellite phones: A security analysis of two satphone standards. In *IEEE Symposium on Security and Privacy*, 2012.
- [19] E. Ekici, I.F. Akyildiz, and M.D. Bender. Datagram routing algorithm for LEO satellite networks. In *Proceedings IEEE INFOCOM. Conference on Computer Communications*, 2000.
- [20] O. Ercetin, S. Krishnamurthy, Son Dao, and L. Tassiulas. A predictive QoS routing scheme for broadband low earth orbit satellite networks. In *IEEE International Symposium on Personal Indoor and Mobile Radio Communications*, 2000.
- [21] S.K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. Bohatei: Flexible and elastic DDoS defense. *USENIX Security Symposium*, pages 817–832, 2015.
- [22] Daniel Fischer, David Basin, Knut Eckstein, and Thomas Engel. Predictable mobile routing for spacecraft networks. *IEEE Transactions on Mobile Computing*, 12(6):1174–1187, 2013.
- [23] Daniel Fischer, David Basin, and Thomas Engel. Topology dynamics and routing for predictable mobile networks. In *IEEE International Conference on Network Protocols*, 2008.
- [24] Y. Gilad, A. Herzberg, M. Sudkovitch, and M. Gberman. CDN-ondemand: An affordable DDoS defense via untrusted clouds. *Network and Distributed System Security Symposium - NDSS*, pages 1–15, 2016.
- [25] Giacomo Giuliani, Tobias Klenze, Markus Legner, David Basin, Adrian Perrig, and Ankit Singla. Internet backbones in space. *ACM SIGCOMM Computer Communication Review*, 50(1):25–37, 2020.
- [26] Deli Gong, Muoi Tran, Shweta Shinde, Hao Jin, Vyas Sekar, Prateek Saxena, and Min Suk Kang. Practical verifiable in-network filtering for DDoS defense. In *IEEE International Conference on Distributed Computing Systems*, 2019.
- [27] Vidyashankar V Gounder, Ravi Prakash, and Hosame Abu-Amara. Routing in LEO-based satellite networks. *IEEE Emerging Technologies Symposium. Wireless Communications and Systems*, pages 22.1–22.6, 1999.
- [28] Mark Handley. Delay is not an option. In *ACM Workshop on Hot Topics in Networks - HotNets*, 2018.
- [29] Mark Handley. Using ground relays for low-latency wide-area routing in megaconstellations. In *ACM Workshop on Hot Topics in Networks - HotNets*, 2019.
- [30] Todd Harrison, Katylin Johnson, and Thomas G. Roberts. Space threat assessment 2018. <https://www.csis.org/analysis/space-threat-assessment-2018>, 2018.
- [31] Min Suk Kang, Soo Bum Lee, and V. D. Gligor. The Crossfire attack. In *IEEE Symposium on Security and Privacy*, 2013.
- [32] T.S. Kelso. Validation of SGP4 and IS-GPS-200D Against GPS Precision Ephemerides. *AAS/AIAA Space Flight Mechanics Conference*, 2007.
- [33] Tobias Klenze, Giacomo Giuliani, Christos Pappas, Adrian Perrig, and David Basin. Networking in Heaven as on Earth. In *ACM Workshop on Hot Topics in Networks - HotNets*, 2018.
- [34] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer-Verlag Berlin Heidelberg, 2012.
- [35] Kuiper Systems LLC. Application of Kuiper Systems LLC for Authority to Launch and Operate a Non-Geostationary Satellite Orbit System in Ka-band Frequencies. https://licensing.fcc.gov/myibfs/download.do?attachment_key=1773885, 2019.
- [36] Aleksandar Kuzmanovic and Edward W. Knightly. Low-rate TCP-targeted denial of service attacks. In *Conference on Applications, technologies, architectures, and protocols for computer communications*, 2003.
- [37] Soo Bum Lee and Virgil D Gligor. FLoc: Dependable link access for legitimate traffic in flooding attacks. In *IEEE International Conference on Distributed Computing Systems*, pages 327–338, 2010.
- [38] Soo Bum Lee, Min Suk Kang, and Virgil D Gligor. CoDef: Collaborative defense against large-scale link-flooding attacks. In *ACM International Conference on emerging Networking EXperiments and Technologies - CoNEXT*, pages 417–428, 2013.
- [39] X. Liu, X. Yang, and Y. Lu. To filter or to authorize: Network-layer DoS defense against multimillion-node botnets. *ACM SIGCOMM Conference on Data Communication*, pages 195–206, 2008.
- [40] Xin Liu, Xiaowei Yang, and Yong Xia. NetFence: preventing internet denial of service from inside out. *ACM SIGCOMM Computer Communication Review*, 40(4):255–266, 2010.

- [41] Z. Liu, H. Jiny, Y.-C. Hu, and M. Bailey. Middlepolice: Toward enforcing destination-defined policies in the middle of the internet. *ACM Conference on Computer and Communications Security*, 24-28-October-2016:1268–1279, 2016.
- [42] R. Mahajan, S.M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. *Computer Communication Review*, 32(3):62–73, 2002.
- [43] S. A. M. Makki, Niki Pissinou, and Philippe Daroux. LEO satellite communication networks - a routing approach. *Wireless Communications and Mobile Computing*, 3(3):385–395, 2003.
- [44] Roland Meier, Petar Tsankov, Vincent Lenders, Laurent Vanbever, and Martin Vechev. Nethide: Secure and practical network topology obfuscation. In *USENIX Security Symposium*, pages 693–709, 2018.
- [45] J. Mirkovic, G. Prier, and P. Reiher. Attacking DDoS at the source. *International Conference on Network Protocols, ICNP*, pages 312–321, 2002.
- [46] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press, 2017.
- [47] Mynaric. Flight terminals (space). <https://mynaric.com/products/space/>, 2020.
- [48] W.D. Nordhaus and X. Chen. Global gridded geographically based economic (G-Econ) data set, version 4. <https://sedac.ciesin.columbia.edu/data/set/spatialecon-gecon-v4>, 2016.
- [49] E.J. Ohlmeyer. Analysis of an ultra-tightly coupled GPS/INS system in jamming. In *IEEE/ION Position, Location, And Navigation Symposium*, 2006.
- [50] James Pavur, Daniel Moser, Martin Strohmeier, Vincent Lenders, and Ivan Martinovic. A tale of sea and sky on the security of maritime VSAT communications. In *IEEE Symposium on Security and Privacy*, 2020.
- [51] Radware. Radware’s DefensePro DDoS Protection. <https://www.radware.com/products/defensepro/>, 2020.
- [52] Barath Raghavan and Alex C Snoeren. A system for authenticated policy-compliant routing. *ACM SIGCOMM Computer Communication Review*, 34(4):167–178, 2004.
- [53] M Rajanna, Shiva Murthy, and Kantharaju H C. Satellite networks routing protocol issues and challenges: A survey. *International Journal of Innovative Research in Computer and Communication Engineering*, 2:153, 2007.
- [54] S. Ramanathan, J. Mirkovic, M. Yu, and Y. Zhang. SENSS against volumetric DDoS attacks. *Annual Computer Security Applications Conference*, pages 266–277, 2018.
- [55] Ryan Rasti, Mukul Murthy, Nicholas Weaver, and Vern Paxson. Temporal lensing and its application in pulsing denial-of-service attacks. In *IEEE Symposium on Security and Privacy*, 2015.
- [56] H. Rausch. Jamming commercial satellite communications during wartime an empirical study. In *IEEE International Workshop on Information Assurance*, pages 8 pp.–118, 2006.
- [57] Desmond Schmidt, Kenneth Radke, Seyit Camtepe, Ernest Foo, and Michał Ren. A survey and analysis of the GNSS spoofing threat and countermeasures. *ACM Computing Surveys*, 48(4):1–31, 2016.
- [58] Ankit Singla, Balakrishnan Chandrasekaran, P. Brighten Godfrey, and Bruce Maggs. The internet at the speed of light. In *ACM Workshop on Hot Topics in Networks, HotNets*, pages 1:1–1:7, 2014.
- [59] SpaceNews. SpaceX adds laser crosslinks to polar Starlink satellites. <https://spacenews.com/spacex-adds-laser-crosslinks-to-polar-starlink-satellites/>, 2021.
- [60] SpaceX. SpaceX Non-Geostationary Satellite System. <https://fcc.report/IBFS/SAT-LOA-20161115-00118/1158350.pdf>, 2016.
- [61] SpaceX. SpaceX Non-Geostationary Satellite System, modification on satellite space station filing. <https://fcc.report/IBFS/SAT-MOD-20190830-00087/1877671>, 2019.
- [62] SpaceX. Application for modification of blanket earth station authorization. <https://fcc.report/IBFS/SES-MOD-INTR2020-02035/2612707>, 2020.
- [63] SpaceX. SpaceX Non-Geostationary Satellite System, modification on satellite space station filing. https://licensing.fcc.gov/myibfs/download.do?attachment_key=2274316, 2020.
- [64] Ahren Studer and Adrian Perrig. The Coremelt attack. In *Computer Security – ESORICS*, pages 37–52, 2009.
- [65] Telesat. Telesat Lightspeed. <https://www.telesat.com/leo-satellites/>, 2021.

- [66] Thales. VasseLINK user manual. <https://www.verasatglobal.com/wp-content/uploads/2019/02/Thales-Vesselink-User-Manual.pdf>, 2020.
- [67] Muhammad Usman, Marwa Qaraqe, Muhammad Rizwan Asghar, and Imran Shafique Ansari. Mitigating distributed denial of service attacks in satellite networks. *Transactions on Emerging Telecommunications Technologies*, 31(6), 2020.
- [68] M. Werner. A dynamic routing concept for ATM-based satellite personal communication networks. *IEEE Journal on Selected Areas in Communications*, 15(8):1636–1648, 1997.
- [69] Lloyd Wood. *Internetworking with satellite constellations*. PhD thesis, University of Surrey, 2001.
- [70] Lloyd Wood. Satellite constellation networks. In *Internetworking and Computing Over Satellite Networks*, pages 13–34. Springer, 2003.
- [71] Yipeng Wu, Zhihua Yang, and Qinyu Zhang. A novel DTN routing algorithm in the GEO-relaying satellite network. In *International Conference on Mobile Ad-hoc and Sensor Networks*, 2015.
- [72] Abraham Yaar, Adrian Perrig, and Dawn Song. SIFF: A stateless Internet flow filter to mitigate DDoS flooding attacks. In *IEEE Symposium on Security and Privacy*, pages 130–143, 2004.
- [73] Yuan Yang, Mingwei Xu, Dan Wang, and Yu Wang. Towards energy-efficient routing in satellite networks. *IEEE Journal on Selected Areas in Communications*, 34(12):3869–3886, 2016.

RainBlock: Faster Transaction Processing in Public Blockchains

Soujanya Ponnappalli¹, Aashaka Shah¹, Souvik Banerjee¹,
Dahlia Malkhi², Amy Tai³, Vijay Chidambaram^{1,3}, and Michael Wei³

¹University of Texas at Austin, ²Diem Association and Novi Financial, ³VMware Research

Abstract

We present RAINBLOCK, a public blockchain that achieves high transaction throughput without modifying the proof-of-work consensus. The chief insight behind RAINBLOCK is that while consensus controls the rate at which new blocks are added to the blockchain, the number of transactions in each block is limited by I/O bottlenecks. Public blockchains like Ethereum keep the number of transactions in each block low so that all participating servers (miners) have enough time to process a block before the next block is created. By removing the I/O bottlenecks in transaction processing, RAINBLOCK allows miners to process more transactions in the same amount of time. RAINBLOCK makes two novel contributions: the RAINBLOCK architecture that removes I/O from the critical path of processing transactions (txs), and the distributed, multi-versioned DSM-TREE data structure that stores the system state efficiently. We evaluate RAINBLOCK using workloads based on public Ethereum traces (including smart contracts). We show that a single RAINBLOCK miner processes 27.4K txs per second (27× higher than a single Ethereum miner). In a geo-distributed setting with four regions spread across three continents, RAINBLOCK miners process 20K txs per second.

1 Introduction

Blockchains maintain the history of transactions as an immutable chain of blocks; each block has an ordered list of transactions, and is processed after its parent or previous block. Blockchains can be public, allowing untrusted servers to process transactions [1,2], or private, allowing only a few specific servers [17]. With their decentralized nature, fault tolerance, transparency, and auditability, public blockchains have led to several applications in a wide range of domains like cryptocurrencies [1,2,19], games [29], and healthcare [43].

In general, public blockchains work in the following manner. Servers participating in the blockchain, termed miners, receive transactions from users. Miners execute the transactions and package them up into blocks. A consensus protocol, like proof-of-work (PoW) [34], decides the next block to be added to the blockchain. With PoW, miners release a new block at a regular cadence (*e.g.*, every 10–12 seconds in Ethereum [2]).

Problem: Low throughput. Public blockchains suffer from low transaction throughput. Two popular public blockchains,

Metric	No state	State: 10M	Ratio
Time taken to mine txs (s)	1047	6340	6× ↑
# Tx per block	2150	833	2.5× ↓
Tx throughput (txs/s)	28.6	4.7	6× ↓

Table 1: Impact of system state on blockchain throughput. This table shows the throughput of Ethereum with proof-of-work consensus when 30K txs are mined using three miners, in two scenarios: first, there are no accounts on the blockchain, and in the second, 10M accounts have been added. Despite no other difference, tx throughput is 6× lower in the second scenario; we trace this to the I/O involved in processing txs.

Bitcoin [1] and Ethereum, process only tens of transactions per second, limiting their applications [33,65].

Prior work traces back the low throughput of blockchains to their proof-of-work (PoW) consensus. PoW limits the block creation rate so that miners have enough time to receive and process the previous block before the next block is created [59]. This ensures that most of the miners are building on the same previous block, preventing forks in the blockchain. Researchers have proposed new consensus protocols [5,30,31,42,46] to increase the transaction throughput.

Insight. We observe that while PoW limits the block creation rate, it *does not* limit the block size (number of transactions in each block). The block size is limited by the rate at which miners can process transactions (§2). Processing transactions involves executing a transaction and modifying the system state accordingly; this becomes more expensive as the state grows. Table 1 experimentally shows that when the number of accounts increases, block size reduces in Ethereum even with PoW consensus (only among three miners). The chief insight in this paper is that *if we could increase the rate at which transactions are processed, we could increase the block size safely, without modifying the PoW consensus protocol.*

How can we increase the block size safely? Miners will continue to release a block every 10–12s after proof-of-work; however, miners can pack more transactions into each block due to faster processing. Typically, increasing the block size increases the time taken to transmit that block, and time taken by miners to process the block. Previously, when Ethereum

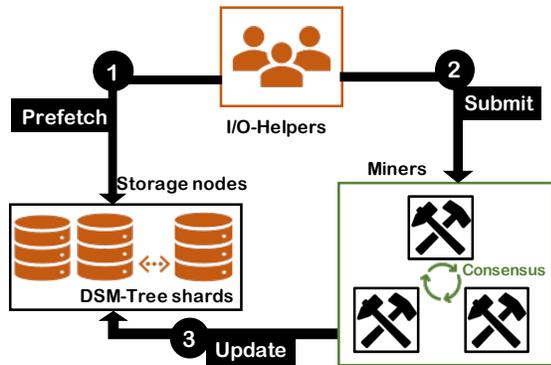


Figure 1: **RAINBLOCK architecture.** I/O-Helpers read data from in-memory storage nodes (out of the critical path) on behalf of the miners. Miners execute txs without performing I/O; storage nodes are updated asynchronously.

increased the maximum block size [21], it was observed that the time taken to propagate the block increased marginally, while the time taken to process transactions saw a significant increase [22]. Thus, if we can increase the rate at which transactions are processed, miners can pack more transactions into each block. Note that as proof-of-work is not modified, the safety properties will continue to hold. The block creation rate, and the transaction confirmation latency will remain the same. However, the rate at which transactions are confirmed will increase as there will be more transactions in each block.

Approach: reducing I/O bottlenecks in tx processing. This work takes the novel approach of increasing tx throughput by tackling I/O bottlenecks in tx processing. Processing a transaction involves executing the transaction and verifying that the execution result is valid. In Ethereum, both execution and verification require reading and writing system state that is stored in a Merkle tree [11,44] on local storage. Tx processing is bottlenecked by I/O: for example, processing a single block of 100 txs in Ethereum requires performing more than 10K random I/O operations (100× higher) and takes hundreds of milliseconds even on a datacenter-grade NVMe SSD (§2).

These I/O bottlenecks arise from two sources. First, the Merkle tree is serialized and stored in a RocksDB [10] key-value store. As a result, traversing the Merkle tree requires multiple, expensive RocksDB reads [53]. Second, Ethereum miners process txs one by one in the critical path; this reduces parallelism and allows I/O bottlenecks to limit the tx throughput. These I/O bottlenecks are an intrinsic part of how Ethereum (and more generally, public blockchains that use Merkle trees) are designed [54]; merely upgrading to faster storage (or even holding all state in memory) will not resolve these problems. By removing these I/O bottlenecks, this work accelerates tx processing, and allows miners to pack more txs in each block, and thereby increases the overall throughput.

This work. This paper presents RAINBLOCK, a new architec-

ture for building public blockchains, that increases the overall throughput with faster transaction processing. RAINBLOCK tackles I/O bottlenecks by designing a custom storage solution for blockchains; at the heart of this storage is a novel data structure, the Distributed, Sharded Merkle Tree (DSM-TREE). The DSM-TREE stores data in a multi-versioned fashion across shards, and allows concurrent reads and writes. DSM-TREE eliminates the inherent serialization from using key-value stores like RocksDB. RAINBLOCK deconstructs miners into three entities: storage nodes that store data, miners that process txs, and I/O-Helpers that fetch data and proofs from storage nodes and provide them to miners, as shown in Figure 1. By having I/O-Helpers prefetch data on behalf of miners, I/O is removed from the critical path of tx processing; moreover, multiple I/O-Helpers can prefetch data at the same time.

Challenges. The RAINBLOCK architecture has to solve several challenges to be effective. For example, Ethereum considered using stateless clients [24] with separate storage nodes; proofs that validate the data sent (termed witnesses) are included in the blocks. The size of these blocks that must be sent over the network made the scheme impractical [25,27], and the proposal was abandoned [57]. RAINBLOCK handles this by co-designing the storage nodes and caches at the miners, minimizing the data sent over the network. A second challenge is prefetching data for Turing-complete smart contracts [56]. As smart contracts can execute arbitrary code, it is not straightforward for the I/O-Helpers to prefetch all the required data. RAINBLOCK solves this by having I/O-Helpers *speculatively pre-execute* the transaction to obtain data and proofs. A third related challenge is that I/O-Helpers may submit stale proofs to the miners (miners may update storage after I/O-Helpers finish reading). RAINBLOCK handles this by having miners tolerate stale proofs whenever possible, via *witness revision*, allowing transactions that would otherwise abort to execute.

Implementation. We implement the RAINBLOCK prototype using Ethereum. We chose Ethereum as our implementation base and comparison point for two reasons. First, Ethereum has been operating as a public blockchain for nearly six years, providing large amounts of data to test our assumptions. Second, Ethereum supports Turing-complete smart contracts, allowing the codification of complex decentralized applications.

Security and Trust. Ethereum has increased the block size in the past [14], without triggering any negative events such as increased forks [16]. RAINBLOCK does the exact same thing, without changing the PoW consensus or block creation rate; as a result, RAINBLOCK inherits the security guarantees of Ethereum. RAINBLOCK introduces storage nodes and I/O-Helpers; however, neither storage nodes, nor I/O-Helpers, nor miners, trust each other. All data exchanged between these entities is authenticated using Merkle trees and can be verified. As a result, the new architecture of RAINBLOCK, while certainly increasing the complexity of the system, does not change its core security or trust assumptions (§3.6).

Evaluation. To evaluate RAINBLOCK, we generate synthetic workloads that mirror transactions on Ethereum mainnet. We analyzed Ethereum transactions and observed that user accounts involved in transactions have a Zipfian distribution: 90% of transactions involve the same 10% of user accounts. We observed that only 10–15% of Ethereum transactions invoke smart contracts. We build a workload generator that faithfully reproduces these distributions. We evaluate RAINBLOCK using these workloads and observe that a single RAINBLOCK miner processes $27\times$ more transactions than an Ethereum miner; we provide a breakdown of the performance difference (§6). When the RAINBLOCK miners are geo-distributed across three continents (thus incurring significant network latency), RAINBLOCK throughput reduces only by 20% compared to a single miner, achieving 20K transactions per second. Since RAINBLOCK has the same proof-of-work consensus and block creation rate as Ethereum, RAINBLOCK finalizes $20\times$ more transactions (with the same latency) as Ethereum. In summary, this paper makes the following contributions:

- The novel RAINBLOCK architecture (§3)
- The novel DSM-TREE authenticated data structure (§4)
- Empirical evidence that throughput in public blockchains can be increased without modifying PoW consensus (§6)

2 Background and Motivation

We provide some background on public blockchains and detail the I/O bottlenecks in transaction processing.

2.1 Public Blockchains and Ethereum

Blockchains are decentralized databases with support for transactions (txs). Blockchains log these txs as a chain of blocks; every block stores an ordered list of txs along with the cryptographic hash of its previous block (parent). Private blockchains allow specific servers to extend the blockchains, while public blockchains allow anyone to participate. As a result, all participating servers in public blockchains are untrusted; any of them can be malicious. Public blockchains depend on the non-malicious majority for correct behavior.

How do public blockchains work? We use Ethereum, a popular public blockchain, as reference. Servers that attempt to add a new block to the blockchain are termed *miners*. Miners receive txs submitted by users, execute these txs, and group them into a block. We term executing txs and grouping into a block as *processing txs*. Several miners compete to add a block to the blockchain. Each miner tries to solve a proof-of-work (PoW) puzzle; the miner that solves the puzzle attaches the solution and broadcasts its block to other miners. Other miners verify the PoW solution and the txs in the block, and build on top of the block. A tx is *confirmed* or *finalized* once ten blocks are built on top of the block containing the tx.

Typically, a miner has two threads as shown in Figure 2. The worker thread executes and groups txs into a partial block.

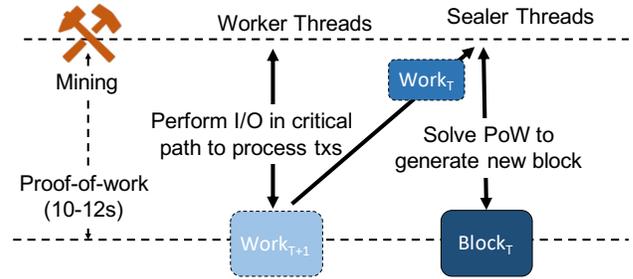


Figure 2: **How Ethereum miners work.** The worker thread processes txs, packages them into a future/partial block, and hands them to the sealer thread. The sealer thread solves the PoW puzzle (in 10–12s), and propose a new block; the worker thread must process txs in this time-frame. I/O bottlenecks result in worker threads packing fewer txs into each block.

The sealer thread obtains the partial block from the worker, and then tries to solve the PoW puzzle; PoW consensus has miners emitting a block every 10–12s, for example. While the sealer thread is working on one block, the worker thread tries to get the next block ready. Thus, the worker has about 10–12s to process txs; if it can process txs faster, it can pack more txs into the partial block that it passes to the sealer.

2.2 Problem: Low throughput

Ethereum and other public blockchains suffer from low throughput: only tens of txs are added to the blockchain per second. The low throughput comes from two factors. First, the PoW consensus limits the block creation rate to one block every 10–12s so that a majority of miners can receive and process a block before a new one is released; this ensures that every miner is building on the same previous block, preventing forks in the blockchain. Note that while PoW consensus limits the block creation rate, it doesn’t directly limit the size of the block. The factor limiting the block size is tx processing time: the rate at which the worker thread can process txs limits the maximum size of the block, as shown in Figure 2.

Tx processing. Processing a tx involves executing the tx and modifying the system state like account values. Since miners do not trust each other in a public blockchain, miners *authenticate data* and can prove that the data they provide is correct. This is done by maintaining a Merkle tree [44] over the data and publishing the latest Merkle root in the blocks; another miner is able to independently execute txs in a block and verify that its local Merkle root matches the root in the block. An account value and its vertical path in the Merkle tree (termed a *witness*) are sufficient to verify the correctness of that value.

Unfortunately, *accessing and authenticating data becomes more expensive as the total size of the data increases* [64]. As a result, tx processing increasingly becomes bottlenecked on I/O. We demonstrate this with an experiment. We create two private Ethereum networks using the Geth client [13];

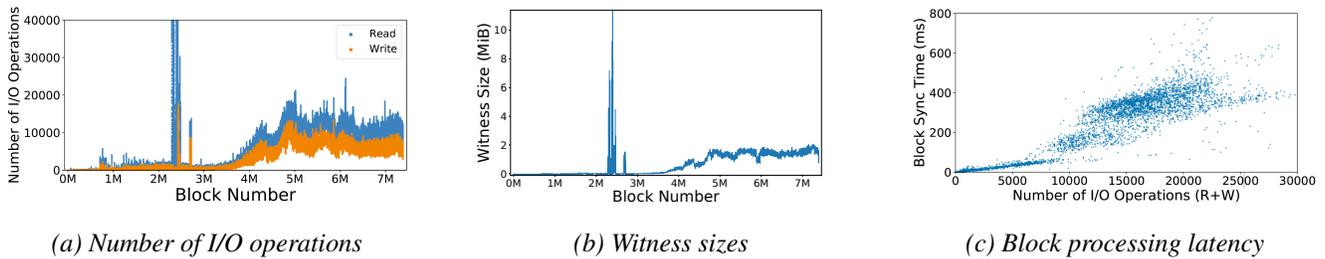


Figure 3: **I/O bottlenecks in processing Ethereum txs.** The increasing system state increases (a) the number of I/O operations performed per block, and (b) the amount of authenticated data accessed (Merkle witnesses) per block. Further, (c) the direct correlation between the block processing time and the number of I/O operations highlights the I/O bottleneck. Spikes in (a) I/O operations and (b) witness sizes are due to a DDOS attack that targeted the system state [64] by creating dummy user accounts.

each network has three miners, 30K txs to mine, and the same proof-of-work (PoW) configurations. While the first network has only 3 miner accounts (total size: 220 MB), the second has 10M additional accounts (total size: 4 GB). Note that Ethereum currently has $\approx 130M$ accounts (total size: > 400 GB [12, 15]) in its blockchain state. Overall, the second network takes $6\times$ more time and $2.5\times$ more blocks to mine all txs using PoW consensus (Table 1). Using profilers, we see that in the second scenario, the time spent solving the PoW puzzle has increased proportionately; however, the worker thread takes $6\times$ more time to process txs, and 69% of this time is spent in accessing and updating the authenticated system state. Thus, miner’s tx processing rate depends on the system state; this impacts the block size and overall throughput. Ethereum uses the modified Merkle Patricia Trie [11] to authenticate the state, we refer to it as the Merkle tree.

Empirical Study. We now study the I/O overheads observed on the Ethereum public blockchain. We use the Parity 2.2.11 client [8] to initialize a new server that joins the Ethereum network, and replays the blockchain (until 7.3 million blocks) to measure various costs. Since we want to observe the historical I/O cost, we do not mine new blocks; rather, we replay the blockchain and execute transactions in blocks that have already been added, termed syncing. The I/O cost remains the same as when mining the block for the first time.

Reading Eth. accounts results in multiple I/O operations. For processing a single block with around 100 transactions, Ethereum performs more than 10K random I/O operations (two orders of difference). Most of these I/O operations are performed for reading and updating user accounts in the Merkle tree. Figure 3(a) shows the number of I/O operations incurred in each block while processing txs, till 7.3 M blocks.

Data authentication causes significant I/O overhead. A witness is the vertical path in the Merkle tree required to verify a data item. Witness sizes represent the amount of data read and modified per block while reading and updating paths in the Merkle tree. In Ethereum, which uses secure 256-bit cryptographic hashes, the witness size of a single 100 byte

user account (or value) can be above 4 KB, resulting in $40\text{--}60\times$ overhead. The witness size also increases as the total data in the Ethereum state increases, as shown in Figure 3(b).

Tx processing is bottlenecked by I/O. We measure the time taken to process (or sync) each block: executing the txs in that block, and verifying if the resultant local Merkle root matches with the Merkle root in that block. Processing an Ethereum block with about 100 txs takes hundreds of milliseconds even on a datacenter-grade NVMe SSD. Figure 3(c) shows the direct correlation between the time taken to process Ethereum blocks and the number of I/O operations performed, indicating that tx processing in Ethereum has I/O bottlenecks.

Summary. The I/O bottlenecks stem from storing the Merkle tree in an LSM-based [49] key-value store like RocksDB [10] that has inherent I/O amplification [52, 53]; Merkle nodes are indexed with their hash, randomizing node locations on disk.

2.3 Straw-man solutions

We consider some straw-man solutions and discuss why they are not suitable for handling I/O bottlenecks in tx processing.

Storing entire system state in memory. Public blockchains like Ethereum do not want to place restrictions on the hardware specifications of miners. If every miner is required to hold the entire authenticated state in memory, only machines with large amounts of DRAM would be able to process transactions. This solution breaks one of the central tenets of Ethereum that contributes to the decentralization of its miners.

Increasing block size. Increasing the block size is the goal of this work; however, doing this naively would not work. If we simply increased the number of txs in each block, miners receiving the block would need more time to process the large block and build on top of it. As a result, the block creation rate would have to be lowered to ensure that previous block is processed by a majority of the miners before a new block is released. Overall, tx throughput would not increase though the block size increased. Over time, I/O bottlenecks exacerbate with the increasing state size, and new servers will take longer to sync and participate in mining. This weakens decen-

tralization. Thus, tackling I/O bottlenecks is crucial to safely increasing block size, and maintaining decentralization.

Alternative consensus protocols. Faster consensus protocols would result in blocks being released quicker, increasing the overall throughput. The goal in this work is to increase throughput *without* changing the consensus, and thus is orthogonal to the work on new consensus protocols. Researchers have noticed that even with faster consensus, blockchains ultimately run into the I/O bottlenecks in tx processing [67].

Thus, we need a mechanism to reduce the I/O bottlenecks in transaction processing. RAINBLOCK achieves this goal with a new architecture and a novel authenticated data structure, the DSM-TREE. With faster transaction processing, RAINBLOCK enables larger blocks and maintains decentralization, *without* changing the PoW consensus or the block creation rate.

3 RainBlock

RAINBLOCK is a public blockchain based on Ethereum that increases overall throughput with faster transaction processing. RAINBLOCK minimizes the I/O bottlenecks in transaction processing, allows miners to safely pack more transactions into each block, and thereby increases the overall throughput.

3.1 Overview

RAINBLOCK minimizes I/O bottlenecks using two techniques. First, it makes each I/O operation cheaper by storing system state in the novel DSM-TREE. In contrast to Ethereum, which accesses data from the RocksDB key-value store on SSDs (where each RocksDB get operation takes between a few hundred microseconds to a few milliseconds), RAINBLOCK stores the system state in memory using the DSM-TREE. DSM-TREE is a sharded, multi-versioned, in-memory authenticated data structure. Second, RAINBLOCK introduces a new architecture that removes I/O from the critical path. RAINBLOCK deconstructs miners into three entities: storage nodes, miners, and I/O-Helpers that read data from storage and submit to miners. In the common case, miners can process txs without performing I/O. Neither miners, nor storage nodes, nor I/O-Helpers trust each other: all data supplied by other entities is verified using Merkle witnesses before use.

RAINBLOCK differs from Ethereum in exactly two aspects: 1) miners are replaced by storage nodes, miners, and I/O-Helpers, and 2) the RocksDB-based storage is replaced with DSM-Tree. Everything else, like the proof-of-work consensus and the block creation rate, remains the same.

3.2 Building up the design step by step

In this section, we start with the problems that our study on Ethereum highlights. We discuss how RAINBLOCK solves these problems and addresses the resulting challenges.

Problem-I: storing authenticated state in key-value stores leads to expensive I/O. Ethereum stores system state in a Merkle tree [11], which is stored in the RocksDB [10] key-value store. Traversing such a Merkle tree requires looking

up nodes using their hashes. Hashing is computationally expensive and results in the nodes of the tree being distributed to random locations on storage. As a result, traversing the Merkle tree to read a leaf value requires several random read operations. Further, the log-structured merge tree [47] that underlies RocksDB results in high I/O amplification [52, 53].

Solution: store state in an optimized in-memory representation. RAINBLOCK introduces an in-memory version of the Merkle tree. Persisting the data is done via a write-ahead log and checkpoints. Traversing the Merkle tree is *decoupled* from hashing; obtaining the next node in the tree is a simple pointer dereference (§4). Note that simply running RocksDB in memory would not be effective, as serializing and hashing Merkle nodes would still add significant overhead.

Resulting challenge: scalability and decentralization. As the blockchain grows, the amount of state in the Merkle tree will increase; soon, a single server's DRAM will not be sufficient. Furthermore, for maintaining decentralization, we cannot require miners participating in the blockchain to have significant amounts of DRAM.

Solution: decouple storage from miners and shard the state. RAINBLOCK solves this problem using separate storage nodes, each of which is a commodity server. RAINBLOCK shards the Merkle tree into subtrees such that each subtree fits in the memory of a storage node. As the amount of data in the state increases, RAINBLOCK increases the number of shards. In this manner, RAINBLOCK scales with commodity miners and storage nodes without reducing the decentralization.

Problem-II: Miners perform slow I/O in the critical path. Transaction processing in Ethereum includes performing slow I/O operations in the critical path, and these transactions are processed one at a time.

Solution: decouple I/O and transaction execution. RAINBLOCK solves this problem by removing the burden of doing I/O from the miners. RAINBLOCK introduces I/O-Helpers that prefetch data and witnesses from the storage nodes and submit them to the miners. Miners use this information to execute transactions without performing I/O and asynchronously update the storage nodes. This architecture also increases parallelism as multiple I/O-Helpers can be prefetching data for different transactions at the same time.

Resulting challenge: Prefetching I/O for smart contracts. One challenge with I/O-Helpers prefetching data is that some transactions invoke smart contracts. Smart contracts are Turing-complete programs that may execute arbitrary code. Thus, how does the I/O-Helper know what data to prefetch?

Solution: pre-execute transactions to get their read and write sets. RAINBLOCK solves this problem by having the I/O-Helpers pre-execute txs. As part of this pre-execution, I/O-Helpers read data and witnesses from the storage nodes. One challenge is that the pre-execution may have different results than when the miner executes the tx (*e.g.*, the smart contract

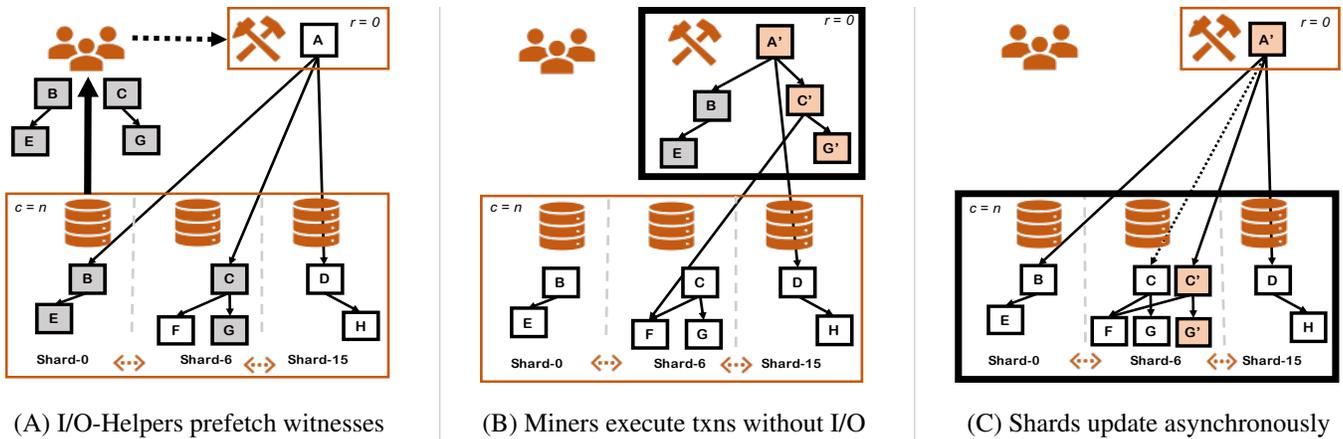


Figure 4: **RAINBLOCK architecture.** This figure shows how RAINBLOCK processes a transaction that reads and updates accounts in two different shards that are along the paths ABE and ACG . (A) I/O-Helpers prefetch witnesses BE and CG from storage nodes and submit them to the miners. (B) Miners verify the witnesses and use them to execute the transaction against their top layer of the DSM-TREE. Then, miners update the storage nodes. (C) Storage nodes verify updates from miners and asynchronously update their bottom layer, creating a new version of the modified Merkle nodes $A'C'G'$. Here, miners retain only the root node of the DSM-TREE ($r = 0$), and the storage nodes send full un-compacted witnesses ($c = n$ or $c = \infty$).

may execute different code based on the block it appears in). We will describe how I/O-Helpers handle smart contracts correctly despite stale data from pre-execution (§3.4).

Resulting challenge: Consistency in the face of concurrency. Another challenge that arises from decoupling I/O from transaction execution is consistency. Multiple I/O-Helpers are reading from the storage nodes, and multiple miners are updating them in parallel. Using locks or other similar mechanisms will reduce concurrency and throughput.

Solution: using the two-layered, multi-versioned DSM-TREE. RAINBLOCK uses DSM-TREE to store the system state. The DSM-TREE has two layers: the *bottom layer* is sharded across the storage nodes and contains multiple versions. Every write causes a new version to be created in a copy-on-write manner; there are no in-place updates. As a result, concurrent updates from miners simply create new versions and do not conflict with each other. Further, each miner uses a DSM-TREE *top layer*: a private, writeable, consistent snapshot of the data, that reflects the miner’s latest version.

Resulting challenge: RAINBLOCK has higher network traffic. Finally, the architecture of RAINBLOCK trades local disk I/O for remote network I/O. As a result, RAINBLOCK results in more network utilization, and the network bandwidth may become the bottleneck.

Solution: RAINBLOCK reduces network I/O via deduplication and the synergy between the DSM-TREE layers. RAINBLOCK uses multiple optimizations to reduce network I/O. First, the bottom and top layer of DSM-TREE collaborate with each other to reduce network traffic. Second, when any component of RAINBLOCK sends witnesses over the network, it

will batch witnesses and perform deduplication to ensure only a single copy of each Merkle tree node is sent. Finally, miners send logical updates to storage nodes rather than physical updates as logical updates are smaller in size.

3.3 Architecture

RAINBLOCK introduces three kinds of participating entities: I/O-Helpers, miners, and storage nodes. Users send txs to I/O-Helpers, which pre-execute these txs and prefetch data and witnesses from storage nodes. Figure 4(a) shows how I/O-Helpers submit txs and the prefetched information to miners. Miners are responsible for creating new blocks of transactions and extending the blockchain; each miner maintains a private DSM-TREE top layer. Figure 4(b) shows how miners use the submitted information to execute these transactions without performing I/O. Finally, miners create a new block, gossip it to other miners, and update the storage nodes. Storage nodes are responsible for maintaining and serving the system state. They use the multi-versioned bottom layer of the DSM-TREE to provide consistent data to I/O-Helpers while handling concurrent updates from miners. Figure 4(c) shows how the miners asynchronously update the bottom layer of the DSM-TREE at the storage nodes.

3.4 Speculative Pre-Execution

I/O-Helpers read all the witnesses required for executing a tx from storage nodes. While this is straight-forward for simple txs, how do I/O-Helpers handle Turing-complete smart contracts that may access arbitrary locations? I/O-Helpers handle this by *speculatively* pre-executing the smart contract to obtain the read and write set.

However, smart contracts can use the timestamp, or block

number of the block in which they appear, during their execution at the miner. These values are not yet known during their pre-execution at the I/O-Helpers. I/O-Helpers speculatively return an estimated value while pre-executing the contract.

Our analysis of Ethereum contracts shows that despite providing estimated values, I/O-Helpers still successfully prefetch the correct witnesses and node bags. For example, the `CryptoKitties mixGenes` function references the current block number and its hash. Since these numbers only affect written values (and not the read set), substituting approximate values does not affect the witnesses that are prefetched.

We observe that I/O-Helpers can pre-execute with *stale* data and still prefetch the correct witnesses. For example, many contracts are *fixed-address* contracts: their behavior depends only on call inputs. To deal with rare *variable-address* contracts, the miner may asynchronously read from storage nodes after the transaction is submitted. Even in these cases, the I/O-Helper will have retrieved some of the correct witnesses required for the tx (e.g., the `to` and `from` accounts).

3.5 Life of a Transaction in RAINBLOCK

We outline the various actions that take place from the time a tx is submitted, to when it becomes a part of the blockchain.

1. I/O-Helper pre-executes the transaction by fetching data and witnesses from the storage nodes
2. I/O-Helper batches and deduplicates Merkle nodes across multiple witnesses and sends these optimized witnesses (termed node bags), txs, and data, to the miner
3. Miner verifies the node bags and advertises them to others.
4. Miner executes the tx using its top layer and the node bags, without any I/O; miner caches all Merkle nodes it reads or revises from these node bags in its top layer
5. Miner, on solving PoW, creates and advertises a new block to other miners; miner also sends the block (with a new Merkle root) and the logical updates to the storage nodes
6. Storage nodes first validate the block (check if PoW solution in the block solves the puzzle), and then persist the logical updates and return successful to the miner.
7. Storage nodes apply the updates asynchronously, and check if their Merkle root matches the root in the block
8. Other miners validate the block and gossip it to others. Then miners execute its txs using node bags, and accept the block by mining new blocks on top of this block
9. Once a majority of the miners receive, validate, and accept the block, the tx becomes part of the blockchain
10. Once ten or more blocks are mined on this block, the tx is confirmed; storage nodes garbage collect the associated versions from the unconfirmed, competing blocks

3.6 Discussion

We note that RAINBLOCK differs from Ethereum in only two aspects: its architecture and its storage. We discuss how these affect trust, incentives, and security, and discuss the trade-offs.

Trust assumptions. RAINBLOCK does not require trust between any of its components. Miners operate without trusting I/O-Helpers or the storage nodes, as miners re-execute transactions and verify the data they receive from I/O-Helpers; I/O-Helpers verify the data they read from storage nodes; and storage nodes verify new blocks and updates from miners.

Incentives. RAINBLOCK shares elements of its architecture with the Ethereum stateless clients proposal [24] that received community support. The central question is how are the storage nodes incentivized to store and serve the latest system state? We propose a model where I/O-Helpers or users pay storage nodes; stateless clients proposal had a similar solution where users pay storage nodes for access to state via state channels. I/O-Helpers or users can always detect if the data served by storage nodes is incorrect or stale, and penalize any malicious shards. We also believe an ecosystem will develop around RAINBLOCK architecture, with commercial entities offering active storage backups; market economics drives these backups to provide and maintain the latest system state with high availability. Note that I/O-Helpers are an optimization and users can prefetch data themselves if required, or pay the I/O-Helpers. Finally, RAINBLOCK miners behave similar to the miners in Ethereum, and are incentivized to process txs via block rewards. Miners are incentivized to broadcast correct updates to storage nodes to aid the acceptance of their fork. Thus, we outline a few ways to incentivize RAINBLOCK components and leave the full solution to future work.

Security. RAINBLOCK provides similar security guarantees as Ethereum, as it does not change the proof-of-work consensus or trust assumptions between participating servers. RAINBLOCK does not impact the block creation rate, as it packs more transactions per block without changing the total time taken to process a block of transactions.

Availability and DDoS attacks. RAINBLOCK decouples storage from miners and has separate storage nodes. RAINBLOCK requires only one replica of each storage shard to be available for making progress. While DDoS attacks can be mounted on storage nodes in RAINBLOCK, they are not new, cannot tamper with the data, and do not impact the correctness of RAINBLOCK; DDoS attacks are also possible on Ethereum and have been successfully executed in the past [60, 64].

Trade-offs. RAINBLOCK introduces new storage nodes and I/O-Helpers; while this adds more complexity into the system, Ethereum was already considering adding storage nodes and stateless clients. Thus, we believe the additional complexity of RAINBLOCK is a good trade-off for its scalability and performance benefits. RAINBLOCK trades off local storage I/O for accessing memory over network, so the network may be

come a bottleneck. RAINBLOCK recognizes this risk and uses multiple techniques like utilizing the memory available at the miners to cache the top layer of the DSM-TREE, and performing witness compaction and deduplication to reduce network traffic (§4.3). Although RAINBLOCK uses extra resources for separate storage nodes, storage nodes are shared across miners, amortizing the costs. I/O-Helpers also use extra resources; however, they are a performance optimization and can execute read-only txs without involving the miners. If users judge I/O-Helpers not useful, users can prefetch from storage nodes, or turn off prefetching, causing miners to perform I/O.

RAINBLOCK adoption. RAINBLOCK can be deployed incrementally in principle. While RAINBLOCK miners rely on storage nodes and I/O-helpers, other miners can use their local RocksDB-based storage, as the two would be compatible.

4 DSM-Tree

RAINBLOCK stores the system state in the DSM-TREE data structure. The Distributed, Sharded Merkle Tree (DSM-TREE) is an in-memory, multi-versioned, sharded, two-layer variant of the Merkle tree. We first present the in-memory representation of the two layers, then describe each layer in detail, and then discuss how the layers collaborate and their trade-offs in different configurations.

In-Memory Representation. DSM-TREE builds the Merkle tree in memory using pointers. Tree traversal is decoupled from hashing and node serializations: traversing the DSM-TREE requires dereferencing pointers to the next Merkle node; in contrast, Ethereum’s Merkle tree reads a Merkle node from RocksDB using its hash, and then deserializes the node to find the cryptographic hash of the next node. DSM-TREE uses periodic checkpoints for persisting the data. The checkpoints are only used to reconstruct the in-memory data structure in case of failures; reads are always served from memory.

Lazy Hash Resolution. When a leaf node in a Merkle tree is updated, hashes of nodes from the leaf to the root need to be recomputed. Recomputing hashes is expensive as nodes have to be serialized before being hashed. DSM-TREE defers this recomputation; on writes, only the leaf nodes are updated; on a subsequent read, all the modified nodes are rehashed exactly once. Thus, lazy hash resolution improves performance significantly by reducing the number of expensive node hashes and RLP (Recursive Length Prefix) serializations [9].

4.1 Bottom Layer

The bottom layer consists of a number of shards. Each shard is a vertical subtree of the Merkle tree, stored in DRAM. The bottom layer supports multiple versions to allow concurrent updates, as shown in Figure 5. The bottom layer has a write-ahead log to persist logical updates.

Multi-versioning. Each write to the bottom layer creates a new logical version of the tree, in a copy-on-write manner. There are no in-place updates. This versioning is required as

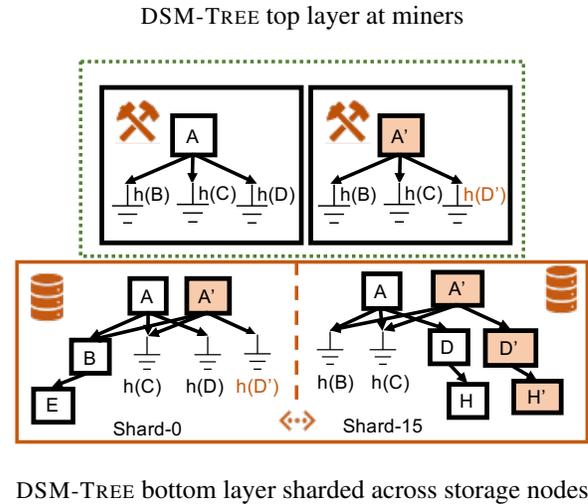


Figure 5: **DSM-TREE design in RAINBLOCK.** This figure shows the two-layered DSM-TREE where miners have their private copy of the top layer for consistency and the bottom layer is sharded for scalability. $h(C)$ is a hash node for C .

miners may submit multiple blocks concurrently that potentially conflict with each other; the bottom layer creates a new version for each write. Thus, writes never conflict with each other, and DSM-TREE does not require locking or additional coordination among miners or I/O-Helpers.

Sharding. Each subtree at the root node of the Merkle tree is a shard; root node in the modified Merkle Patricia Trie has 16 children, so DSM-TREE has 16 shards by default. Moreover, individual subtrees in each shard can further be partitioned, allowing shards to increase with the increasing system state.

Garbage collection. Garbage collection of versions is driven by the higher-level blockchain semantics. When multiple miners are working on competing forks of the blockchain, multiple versions are maintained. Eventually, one of the forks is confirmed and accepted as the mainline fork, and the others are discarded; associated versions from discarded blocks are garbage collected by the bottom layer of the DSM-TREE.

4.2 Top Layer

Given that the bottom layer maintains multiple versions across multiple shards, what data should be read by the miners? To resolve this question, miners use the top layer. Each miner has a private top layer, that represents a consistent, writeable snapshot of the system state; it contains the first few levels of the Merkle tree, till a configurable retention level (r), and has the Merkle root node that summarizes a consistent snapshot of the entire system state. When a miner executes transactions, all reads return values from this snapshot of the system.

As a miner executes txs, their top layer is updated, switching to another consistent view of the state as shown in Figure 5. The miner asynchronously updates the bottom layer’s shards.

Caching and Pruning. The top layer acts as an in-memory cache of witnesses for the miner. By design, the top layer stores the recently used and the frequently changing parts of the Merkle tree. When the miner receives witness from I/O-Helpers, the top layer uses the witnesses to reconstruct a *partial* Merkle tree that allows miners to execute transactions, typically without performing I/O. The top layer also supports pruning the partial Merkle tree to help miners reclaim memory. Pruning replaces the nodes at the retention level ($r + 1$) with *Hash nodes*. Hash nodes are placeholders that help miners to identify the DSM-TREE shard which has the pruned nodes.

Witness Revision. The bottom layer of the DSM-TREE is updated asynchronously by miners. As a result, the top layer (miner) may receive stale witnesses from the bottom layer (via I/O-Helpers). We introduce a new technique termed *witness revision* to tolerate stale witnesses. A witness is determined to be stale or incorrect because the Merkle root in the witness doesn't match the top layer's Merkle root. However, this could happen because of an *unrelated update* to another part of the Merkle tree. For example, the top layer might contain one vertical path; a different vertical path might have been updated. The top layer detects when this happens, and *revises* the witness to make it current by applying updates to the witness that are known to the miner. If the Merkle root matches now, then the witness is accepted. Witness revision is similar to doing `git push` (trying to upload your changes), finding out something else in the repository has changed, doing a `git pull` (obtaining the changes in the repository) to merge changes, and then doing a `git push`. With witness revision, the top layer tolerates stale data from the bottom layer and allows miners to execute non-conflicting transactions that would otherwise get rejected. Note that, witness revision cannot revise every potential stale witnesses. If the top layers are pruned aggressively, they may have insufficient information to detect if the changes are from an unrelated part of the Merkle tree.

4.3 Synergy between the layers

The top and bottom layers collaborate to reduce network traffic. We also briefly discuss the potential DSM-TREE configurations with r (retention at the top layer) and c (compaction level at the bottom layer), and the tradeoffs involved.

Witness compaction. As the top layer of the DSM-TREE stores the top levels of the Merkle tree, the storage nodes need not send a full witness. Like the configurable retention level at the top layer r , the bottom layer has a configurable compaction level, c . If witnesses are larger than c , witnesses are compacted by removing the top few Merkle nodes, and are sent over the network. In addition, multiple witnesses are batched and Merkle nodes are deduplicated (node bagging), further reducing the network burden of transmitting witnesses.

Configurations. The DSM-TREE can be configured to operate entirely from local memory without any network overhead, or just from remote memory with high network utilization. For

example, if the top layer of the DSM-TREE has $r = \infty$, then the top layer caches the entire Merkle tree and is fully served from local memory. Similarly, if the bottom layer has $c = n$ or $c = \infty$, then un-compacted witnesses are sent over the network and accessed entirely from remote memory, as shown in the Figure 4. These parameters (r and c) can be tuned based on the available memory and network capacity.

Tradeoffs. In a Merkle tree that has n levels, any DSM-TREE configuration that satisfies $c \geq (n - r)$ allows the top layer to use compacted witnesses from the bottom layer. Note that having a higher r results in a lower number of transaction aborts, as the top layer has more information to detect non-conflicting updates and perform witness revision. The top layers should therefore set r based on the amount of memory available. Pruning the top layer should only be done under memory pressure.

4.4 Summary

The DSM-TREE is a novel variant of the Merkle tree, modified for faster transaction processing in public blockchains. DSM-TREE presents a new point in the design space of authenticated data structures. DSM-TREE highlights the benefits of in-memory pointer-based tree traversals in comparison to using node hashes and RocksDB lookups. The top layer exploits the cache-friendliness of the Merkle tree (top nodes are frequently read and updated), while the sharded bottom layer relies on the fact that witness creation only requires a vertical slice of the tree. DSM-TREE top layer uses witness revision to handle stale data from concurrent updates. While the DSM-TREE supports transactions and is exclusively used with RAINBLOCK in this paper, it can be easily modified to work with other blockchains and applications.

5 Implementation

We implement RAINBLOCK and DSM-TREE in Typescript, targeting `node.js`. Miners and storage nodes use the DSM-TREE library¹. The performance critical portions of the code, such as `secp256k1` key functions for signing transactions and generating `keccak` hashes, are written as C++ `node.js` bindings. To execute smart contracts, we implement bindings for the Ethereum Virtual Machine Connector interface (EVMC) and use Hera (v0.2.2). Hera can run contracts implemented using Ethereum flavored WebAssembly (ewasm) or EVMC1 bytecode through transcompilation. The I/O-Helper is implemented in C++. DSM-TREE and RAINBLOCK, together 15K lines of code, are open source and available on Github². RAINBLOCK assumes 16 shards by default; this is configurable.

6 Evaluation

We seek to answer the following questions:

- What is the performance of a single miner? (§6.1)

¹www.npmjs.com/package/@rainblock/merkle-patricia-tree

²<https://github.com/RainBlock>

System/optimization	Get	Put
In-memory Ethereum MPT	1x	1x
Pointer-based traversal	2.7x	2.3x
Batching and Lazy hash resolution	56x	69x
All optimizations	150x	160x

Table 2: **Performance breakdown.** The table shows the throughput of DSM-TREE (relative to in-memory Ethereum merkle tree) on gets and puts with different optimizations.

- What is the end-to-end performance of RAINBLOCK in a geo-distributed setting? (§6.2)
- How is performance affected by tunable parameters? (§6.3)
- What are the overheads of RAINBLOCK? (§6.4)

Our technical report [50] contains more details for these experiments, along with additional experimental results.

Experimental setup. We run the experiments in a cloud environment on instances which are similar to the `m4.2xlarge` instance available on Amazon EC2 with 32GB of RAM and 48 threads per node. We use Ubuntu 18.04.02 LTS, and `node.js` v11.14.0. For the end-to-end benchmarks, each storage node, miner, and I/O-Helper is deployed on its own instance.

Workloads. We evaluate the performance of RAINBLOCK against synthetically generated workloads that mirror transactions on the Ethereum public *mainnet* blockchain. Since Ethereum transactions are signed, they cannot be used in experiments: we cannot change transaction data or the source accounts, because we do not have the `secp256k1` private key. To tackle this challenge, we analyze the public blockchain to extract salient features, and develop a *synthetic workload generator* which generates accounts with private keys we control so our I/O-Helpers can run and submit signed transactions.

Synthetic Workload Generator. We analyze the transactions in the Ethereum *mainnet* blockchain to build a synthetic workload generator. We analyzed 100K recent (since block 7M) and 100K older blocks (between blocks 4M and 5M) in the Ethereum blockchain to determine: 1) the distribution of accounts involved in transactions, and 2) the fraction of all transactions that invoke smart contracts. We observe that 10-15% of Ethereum transactions are contract calls and the rest are simple transactions. This is true of both recent blocks and older blocks. It is also the case that a small percentage of accounts are involved in most of the transactions. Based on the analyzed data, we generate workloads where 90% of accounts are called 10% of the time, and 10% of the accounts are called 90% of the time. Smart contracts are invoked 15% of the time.

6.1 Performance of a single miner

The performance of a single miner depends on three things: the DSM-TREE data structure, the I/O-Helpers, and the top-

layer cache at the miner. We first evaluate the performance of the DSM-TREE data structure, and then measure overall tx processing performance on a single node varying the number of I/O-Helpers. We then show the performance of the miner when varying the retention level of the top-layer cache.

DSM-TREE performance. For a fair comparison, we configure Ethereum to use an in-memory key-value store for storage. Ethereum uses the in-memory Merkle Patricia-Trie (MPT) [6] implemented using the `memdown` red-black tree [7]. We use `put` operations to recreate the system state corresponding to four million blocks on the Ethereum public chain. This results in 1.19M accounts. We read all accounts sequentially using `get` operations. `Put` operation creates or updates user account with 160-bit Ethereum address; `get` operation returns the RLP-encoded [9] Ethereum account at the address, along with its witness containing RLP-encoded Merkle nodes. We also compare the memory used.

Performance breakdown. DSM-TREE outperforms Ethereum’s in-memory Merkle tree significantly in both gets and puts, as shown in Table 2. Note that both data structures are in memory, so the performance difference comes from other optimizations. Eth MPT has to use node hashes to traverse the Merkle Tree; in contrast, DSM-TREE uses pointers for constructing the tree, and eliminates hashing. This feature improves performance by 2.7x. Eth MPT has to hash and serialize, or deserialize each node in the Merkle tree while writing or reading them; DSM-TREE optimizes this with memoization and batching. Memoization allows remembering the hashes and RLP-encodings of unmodified Merkle nodes. Memoizing RLP-encoded nodes in DSM-TREE increases `get` performance by reducing redundant node de-serializations. Further, with batching, common nodes in the upper part of the Merkle Tree are only deserialized once, increasing the `get` performance by 56x relative to MPT, bringing the overall performance difference between DSM-TREE and Eth MPT to be 150x. A good overall intuition for these performance improvements is the difference between a linked list in memory versus a linked list where each node is serialized and stored in an in-memory key-value store using the node’s hash as its key.

The performance difference for puts is similar (160x). The efficient memory representation of DSM-TREE contributes to 2.3x of this performance difference; the rest of the difference is due to lazy hash resolution. Lazy hash resolution defers recomputing the hashes of inner tree nodes until they are read. As a result, only the leaf nodes are updated in the critical path. If we force all the Merkle nodes to be updated and rehashed after every thousand updates, the performance difference with Eth MPT drops to 5x overall for puts.

Memory consumption. For the same system state with 1.19M accounts, DSM-TREE consumes 34x lower memory (775 MB) than Ethereum MPT (25 GB). This results from Eth MPT storing each node as a key-value pair. The reduced memory consumption of DSM-TREE is important since we

Optimizations	Config	Txs/s
Baseline	Ethereum, 1 miner	1K (1×)
RAINBLOCK	1 miner, 1 helper, $r=0$	2.6 K (2.6×)
Prefetch in parallel	1 miner; 4 helpers, $r=0$	7.7K (7×)
DSM-TREE tuning	1 miner; 4 helpers, $r=7$	27.4 K (27×)
Geo-distributed	4 miners; 16 helpers, $r=8$	20K (20×)

Table 3: **Performance breakdown.** The table shows the throughput of RAINBLOCK with different optimizations. All configs use 16 storage shards. Helpers indicate I/O-Helpers. While parallel prefetching increases RAINBLOCK throughput by 2.9× from 2.6K to 7.7K tps, the DSM-TREE top layer caching and witness compaction further increase throughput by 3.5×, from 7.7K to 27.4K tps.

want each shard to fit in the memory of a commodity server.

Overall performance. An Ethereum miner can process 1000 txs per second, if its system state is stored in an in-memory key-value store. We measured the performance of a single RAINBLOCK miner with one I/O-Helper when the top-layer caching is disabled; the miner is accessing system state from remote in-memory shards. In this setting, RAINBLOCK processes 2600 txs per second (2.6× higher than Ethereum).

Performance with multiple I/O-Helpers. Increasing the number of I/O-Helpers, increases the performance of RAINBLOCK, till up to four I/O-Helpers per miner. With four I/O-Helpers, parallel prefetching increases performance to 7700 txs per second (2.9× higher). Thus, RAINBLOCK miner (with four I/O-Helpers) outperforms an Ethereum miner by 7.7×.

Performance with DSM-TREE tuning. When we configure the top layer of the DSM-TREE to retain the first seven levels ($r = 7$, $c = n - 7$), the miner can process 27400 tps (3.5× higher than when top layer caching and witness compaction was disabled, and 27× higher than a single Ethereum miner).

6.2 End-to-End Geo-distributed Experiment

We run a geo-distributed experiment, with varying numbers of regions across three continents. Each region has four I/O-Helpers, one miner, and 16 storage nodes, caching eight levels of the DSM-TREE tree ($r = 8$, $c = n - 8$).

RAINBLOCK in a single region has a throughput of 25000 txs per second; this is slightly reduced from the 27400 tps in the previous section since the storage nodes are being accessed over a wide-area network. When we scale to four regions, the throughput drops to 20000 txs per second, thus retaining 80% of the single-region performance. When we ran a workload consisting purely of smart contracts (OmiseGO Token), RAINBLOCK achieved 17900 tps. Table 3 captures RAINBLOCK performance in various settings, from a single miner with one I/O-Helper to the geo-distributed experiment.

Tx confirmation latency remains the same as in Ethereum,

as RAINBLOCK and Ethereum share the same block creation and confirmation logic (confirmed after ten blocks build on the block containing the tx). As more txs are present in each block, more txs are confirmed per second. These experiments use the same PoW consensus in Ethereum, thus demonstrating that RAINBLOCK achieves higher tx throughput without modifying the consensus protocol.

6.3 Perf impact of tunable parameters

We discuss the impact of varying two configuration parameters: the retention level r (number of Merkle tree levels stored at the top layer), and the compaction level c (bottom c levels of the witnesses are sent over the network by the shards).

Impact of tuning retention level. Increasing the retention level at miners increases overall tx throughput, but also increases the memory requirements at the miners. Pruning the cache to a certain retention level (r) helps reclaim the memory consumed by miners. For the Merkle tree constructed in the geo-distributed experiment described previously, miners caching till a tree depth of five ($r = 5$) consumes only 40% of the memory consumed by storing the full DSM-TREE in memory. As we increase the number of I/O-Helpers, the impact of higher r decreases. For example, in the geo-distributed experiment in a single region, there was no performance difference between $r = 7$ and $r = 8$ with four I/O-Helpers, but performance improved by 35% between $r = 7$ and $r = 8$ with two I/O-Helpers.

Retention Level and Tx Abort rate. If the top layer doesn't cache enough levels, stale witnesses cannot be revised, leading to tx aborts. We evaluate RAINBLOCK with 16 storage shards, 1 miner, and 4 clients, with various r and c configurations to measure the transaction abort rate. Increasing r reduces the transaction abort rate. Further, when there are a large number of accounts, the contention on Merkle tree nodes reduces, increasing the number of witnesses that can be revised and thus, reducing the abort rate for a fixed r . With 1M accounts and $r=6$, RAINBLOCK aborts less than two txs per second. In cases where the txs get aborted, I/O-Helpers or users themselves can fetch up-to-date witnesses from storage nodes and resubmit transactions to miners.

Tuning compaction level. A lower compaction level c increases the DSM-TREE shard throughput for I/O-Helpers (as it reduces the size of witnesses transmitted over network); with 10M accounts in state and $c = n$, shards process account reads at 1.36K ops/sec and with $c = n - 6$, they process 9.4K ops/sec (7× increase in throughput per shard). The compaction level should be tuned alongside the retention level, with $c \geq (n - r)$ for a Merkle tree with n levels.

6.4 Overheads

RAINBLOCK has two main sources of overhead. First, it trades local storage I/O for network I/O, hence resulting in more network traffic. Second, it requires the participation of more

commodity servers as I/O-Helpers and storage nodes. We discuss these overheads and how RAINBLOCK mitigates them.

Network bandwidth requirements. RAINBLOCK can be configured to produce blocks of a given size. For example, if the network can only handle 1 MB blocks, RAINBLOCK can be configured to produce blocks of this size. In our experiments, we do not constrain RAINBLOCK, and see that RAINBLOCK can pack about 240K transactions into each block ($480\times$ higher than Ethereum), on average, with the same proof-of-work consensus and block creation time. RAINBLOCK blocks are about 24MB in size, compared to the recent Ethereum blocks that are 40-60 KB. Our geo-distributed experiment used 24 MB blocks over the wide-area-network without running into network bottlenecks. The second source of network traffic is gossiping witnesses between miners. Using witness compaction, and node bagging (batching and deduplication), RAINBLOCK reduces witness sizes by 95%, allowing miners to advertise witnesses with commodity network bandwidths. Our witnesses sent over the network were a few KB in size.

Additional resources. RAINBLOCK requires I/O-Helpers and storage nodes. While storage nodes keep all state in memory, the state is sharded so that each shard fits in the DRAM of a commodity server. Storage nodes are shared among all miners, and hence they do not significantly increase the overall cluster requirements; I/O-Helpers can also be shared by miners.

7 Related Work

In this section, we place our contributions RAINBLOCK and DSM-TREE in the context of prior research.

Stateless Clients. The Stateless Clients [24] proposal seeks to insert witnesses into blocks, allowing miners to process a block without I/O. Despite active discussions [3, 25, 27], this proposal has not been implemented due to large witness sizes [57]; a single, simple transaction can have 4-6KB witness sizes, resulting in $40\text{-}60\times$ the network overhead. In contrast, DSM-TREE reduces witness sizes by 95%; RAINBLOCK does not insert witnesses in blocks, and uses I/O-Helpers to reduce the I/O burden on miners.

Hyperledger Fabric. Fabric [17] is private while RAINBLOCK is public, resulting in significant differences. Fabric introduces a new execute-order-validate architecture where txs are executed only on a subset of servers. In RAINBLOCK all miners execute every transaction. While Fabric relies on signatures from trusted nodes (which can become the bottleneck), RAINBLOCK uses witnesses from untrusted servers to authenticate data. Peers in Fabric store the entire state, while RAINBLOCK storage nodes store partitions of the system state. RAINBLOCK improves performance with I/O-efficient transaction processing, while Fabric derives high performance from optimistic execution and efficient consensus.

Sharding. Sharding the blockchain into independent parallel chains that operate on subsets of state [36, 39, 41, 61, 62, 70]

reduces I/O overheads; however, requires syncing the independent chains for consistency, is less resilient to failures or attacks [51, 55, 69], and require complex cross-shard transactions protocols. In contrast, RAINBLOCK does not shard the blockchain; the storage is sharded, but all miners add to a single chain. RAINBLOCK does not require locking or additional communication for executing transactions across multiple storage shards. Payment channels [4, 32, 35, 38, 40, 45] that offload work to side chains are complementary to our work.

Dynamic accumulators. Merkle trees belong to a general family of dynamic accumulators [20, 26]. Merkle trees, allow fast processing but, proofs grow with the underlying state. Constant-size dynamic accumulators based on RSA signatures [20, 26] have fixed size proofs but, have low processing rates; improving their performance is an ongoing effort [23]. DSM-TREE provides a practical solution to achieve high processing rates and small witness sizes. Recent work has proposed many new authenticated data structures [18, 28, 37, 53, 54, 58, 66, 68]. In contrast to these works, DSM-TREE scales Ethereum's Merkle Patricia trie [11] without changing its core structure, or how proofs are generated.

Transaction execution. RAINBLOCK adopts a design similar to Solar [71] and vCorfu [63], where transactions are executed based on data from sharded storage. RAINBLOCK modifies the design for decentralized applications and authenticated data structures. This allows RAINBLOCK to execute transactions on sharded state without requiring locking or additional coordination among miners. Similar to RAMCloud [48], the DSM-TREE design argues that large random-access data structures can get higher throughput and scalability when served from memory over the network.

8 Conclusion

We have presented RAINBLOCK, a public blockchain architecture that increases transaction throughput without changing the proof-of-work consensus protocol. RAINBLOCK achieves this by tackling the I/O bottleneck in transaction processing, allowing miners to pack more transactions into each block. RAINBLOCK introduces a novel architecture that moves I/O off the critical path, and the DSM-TREE, a new authenticated data structure that provides cheap access to system state. Please refer to our technical report [50] for more details about RAINBLOCK and the DSM-TREE. The RAINBLOCK prototype is publicly available at <https://github.com/RainBlock> and we welcome working with the community on its adoption.

Acknowledgements

We thank our shepherd, Abhinav Duggal, and the anonymous reviewers at ATC'21, VLDB'21, NSDI'20, and SOSP'19 for their insightful comments and suggestions. This work was supported by NSF CAREER #1751277, and donations from VMware, Google, and Facebook.

References

- [1] Bitcoin. <https://bitcoin.org/en/>, 2019.
- [2] Ethereum. <https://github.com/ethereum/>, 2019.
- [3] Ethereum improvement proposals repository. <https://github.com/ethereum/EIPs>, 2019.
- [4] Fast, cheap, scalable token transfers for ethereum. <https://raiden.network/>, 2019.
- [5] Hybrid casper ffg. <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1011.md>, 2019.
- [6] Implementation of the modified merkle patricia tree as specified in the Ethereum's yellow paper. <https://github.com/ethereumjs/merkle-patricia-tree>, 2019.
- [7] In-memory abstract-leveldown store for node.js and browsers. <https://github.com/Level/memdown>, 2019.
- [8] Parity ethereum 2.2.11-stable. <https://github.com/paritytech/parity-ethereum/releases/tag/v2.2.11>, 2019.
- [9] Recursive Length Prefix Encoding. <https://github.com/ethereum/wiki/wiki/RLP>, 2019.
- [10] RocksDB | A persistent key-value store. <http://rocksdb.org>, 2019.
- [11] The modified Merkle Patricia tree. <https://github.com/ethereum/wiki/wiki/Patricia-Tree>, 2019.
- [12] Full Node sync with Default Settings. <https://etherscan.io/chartsync/chaindefault>, 2020.
- [13] Geth ethereum 1.9.25-stable. <https://github.com/ethereum/go-ethereum/tree/v1.9.25>, 2020.
- [14] Ethereum average gas limit chart. <https://etherscan.io/chart/gaslimit>, 2021.
- [15] Number of unique addresses in ethereum. <https://etherscan.io/chart/address>, 2021.
- [16] Uncles per day. daily count of uncles generated by the ethereum network. <https://www.etherchain.org/charts/unclesPerDay>, 2021.
- [17] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.
- [18] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. {SPEICHER}: Securing lsm-based key-value stores using shielded execution. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 173–190, 2019.
- [19] BCNext. The nxt cryptocurrency. <https://nxt.org>, November, 2013.
- [20] Josh Benaloh and Michael De Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Workshop on the Theory and Application of Cryptographic Techniques*, pages 274–285. Springer, 1993.
- [21] The block. Ethereum miners are increasing the network's gas limit by 25 = <https://www.theblockcrypto.com/linkedin/69053/ethereum-miners-vote-for-25-gas-limit-increase>, June 20, 2020.
- [22] bloXroute Labs. Increasing eth's gas limit: What we can safely do today. = <https://ethresear.ch/t/increasing-eth-s-gas-limit-what-we-can-safely-do-today/8121>, October 2020.
- [23] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Annual International Cryptology Conference*, pages 561–586. Springer, 2019.
- [24] Vitalik Buterin. The Stateless Clients Concept. <https://ethresear.ch/t/the-stateless-client-concept/172>, 2017.
- [25] Vitalik Buterin. Detailed analysis of stateless client witness size, and gains from batching and multi-state roots. <https://ethresear.ch/t/detailed-analysis-of-stateless-client-witness-size-and-gains-from-batching-and-multi-state-roots/862>, 2019.
- [26] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Annual International Cryptology Conference*, pages 61–76. Springer, 2002.
- [27] Alexander Chepurnoy. A possible solution to stateless clients. <https://ethresear.ch/t/a-possible-solution-to-stateless-clients/4094>, 2019.
- [28] Alexander Chepurnoy, Charalampos Papamanthou, and Yupeng Zhang. Edrax: A cryptocurrency with stateless transaction validation. 2018.
- [29] Tonya M Evans. Cryptokitties, cryptography, and copyright. *AIPLA QUARTERLY JOURNAL*, 47(2):219, 2019.

- [30] Ittay Eyal, Adem Efe Gencer, Emin Gun Sirer, and Robert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 45–59, Santa Clara, CA, 2016. USENIX Association.
- [31] 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*, pages 51–68, New York, NY, USA, 2017. ACM.
- [32] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 473–489, 2017.
- [33] Zane Huffman. CryptoKitties is Clogging the Ethereum Network. <https://themerikle.com/cryptokitties-is-clogging-the-ethereum-network/>, 2019.
- [34] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Communications and Multimedia Security*, 1999.
- [35] Thaddeus Dryja Joseph Poon. The bitcoin lightning network: Scalable off-chain instant payments. <https://lightning.network/lightning-network-paper.pdf>, 2019.
- [36] Vitalik Buterin Joseph Poon. Plasma: Scalable autonomous smart contracts. <https://plasma.io/plasma.pdf>, 2019.
- [37] Janakirama Kalidhindi, Alex Kazorian, Aneesh Khera, and Cibi Pari. Angela: A sparse, distributed, and highly concurrent merkle tree. 2018.
- [38] Rami Khalil and Arthur Gervais. Revive: Rebalancing off-blockchain payment networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 439–453, 2017.
- [39] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. Omniledger: A secure, scalable, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598, May 2018.
- [40] Marta Lohava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafal Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 80–96, 2019.
- [41] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, pages 17–30, New York, NY, USA, 2016. ACM.
- [42] Loi Luu, Yaron Velner, Jason Teutsch, and Prateek Saxena. Smartpool: Practical decentralized pooled mining. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1409–1426, Vancouver, BC, 2017. USENIX Association.
- [43] Thomas McGhin, Kim-Kwang Raymond Choo, Charles Zhechao Liu, and Debiao He. Blockchain in healthcare applications: Research challenges and opportunities. *Journal of Network and Computer Applications*, 135:62–75, 2019.
- [44] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.
- [45] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. Sprites: Payment channels that go faster than lightning. *CoRR abs/1702.05812*, 306, 2017.
- [46] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, pages 31–42, New York, NY, USA, 2016. ACM.
- [47] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.
- [48] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, 2015.
- [49] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.
- [50] Soujanya Ponnappalli, Aashaka Shah, Amy Tai, Souvik Banerjee, Vijay Chidambaram, Dahlia Malkhi, and Michael Wei. Rainblock: Faster transaction processing in public blockchains, 2020.
- [51] Tayebah Rajab, Mohammad Hossein Manshaei, Mohammad Dakhilalian, Murtuza Jadhwal, and Mohammad Ashiqur Rahman. On the feasibility of sybil attacks in shard-based permissionless blockchains. *arXiv preprint arXiv:2002.06531*, 2020.

- [52] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.
- [53] Pandian Raju, Soujanya Ponnappalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. mLSM: Making Authenticated Storage Faster in Ethereum. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, 2018. USENIX Association.
- [54] Leonid Reyzin, Dmitry Meshkov, Alexander Chepurnoy, and Sasha Ivanov. Improving authenticated dynamic dictionaries, with applications to cryptocurrencies. In *International Conference on Financial Cryptography and Data Security*, pages 376–392. Springer, 2017.
- [55] Alberto Sonnino, Shehar Bano, Mustafa Al-Bassam, and George Danezis. Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers. *arXiv preprint arXiv:1901.11218*, 2019.
- [56] Nick Szabo. Smart contracts. *Unpublished manuscript*, 1994.
- [57] Peter Szilagyi. Are stateless clients a dead end? https://www.reddit.com/r/ethereum/comments/e8ujfy/are_stateless_clients_a_dead_end/, December, 10, 2019.
- [58] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. *IACR Cryptol. ePrint Arch.*, 2020:527, 2020.
- [59] Vitalik Buterin. Toward a 12-second Block Time. <https://blog.ethereum.org/2014/07/11/toward-a-12-second-block-time/>, 2014.
- [60] Vitalik Buterin. Transaction spam attack: Next Steps. <https://blog.ethereum.org/2016/09/22/transaction-spam-attack-next-steps/>, 2016.
- [61] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *International workshop on open problems in network security*, pages 112–125. Springer, 2015.
- [62] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 95–112, Boston, MA, February 2019. USENIX Association.
- [63] Michael Wei, Amy Tai, Christopher J Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritch, Steven Swanson, et al. vcorfu: A cloud-scale object store on a shared log. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 35–49, 2017.
- [64] Jeffrey Wilcke. The Ethereum network is currently undergoing a DoS attack. <https://ethereum.github.io/blog/2016/09/22/ethereum-network-currently-undergoing-dos-attack/>, 2016.
- [65] JI Wong. Cryptokitties is causing ethereum network congestion (2017).
- [66] Cheng Xu, Ce Zhang, and Jianliang Xu. vchain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the 2019 international conference on management of data*, pages 141–158, 2019.
- [67] Lei Yang, Vivek Bagaria, Gerui Wang, Mohammad Alizadeh, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Scaling bitcoin by 10,000 x. *arXiv preprint arXiv:1909.11261*, 2019.
- [68] Mingchao Yu, Saeid Sahraei, Songze Li, Salman Avestimehr, Sreeram Kannan, and Pramod Viswanath. Coded merkle tree: Solving data availability attacks in blockchains. *arXiv preprint arXiv:1910.01247*, 2019.
- [69] Jusik Yun, Yunyeong Goh, and Jong-Moon Chung. Trust-based shard distribution scheme for fault-tolerant shard blockchain networks. *IEEE Access*, 7:135164–135175, 2019.
- [70] 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, CCS '18*, pages 931–948, New York, NY, USA, 2018. ACM.
- [71] Tao Zhu, Zhuoyue Zhao, Feifei Li, Weining Qian, Aoying Zhou, Dong Xie, Ryan Stutsman, Haining Li, and Huiqi Hu. Solar: towards a shared-everything database on distributed log-structured storage. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 795–807, 2018.

An Off-The-Chain Execution Environment for Scalable Testing and Profiling of Smart Contracts

Yeonsoo Kim[†], Seongho Jeong[†], Kamil Jezek[‡], Bernd Burgstaller[†], and Bernhard Scholz[‡]
[†]*Yonsei University* [‡]*The University of Sydney*

Abstract

Smart contracts in Ethereum are executable programs deployed on the blockchain, which require a client for their execution. When a client executes a smart contract, a world state containing contract storage and account details is changed in a consistent fashion. Hence, the execution of smart contracts must be sequential to ensure a deterministic representation of the world state. Due to recent growth, the world state has been bloated, making testing and profiling of Ethereum transactions at scale very difficult.

In this work, we introduce a novel off-the-chain execution environment for scalable testing and profiling of smart contracts. We disconnect transactions from the world state by using substates to execute the transactions in isolation and in parallel. Compared to an Ethereum client, our execution environment reduces the space required to replay the transactions of the initial 9 M blocks from 700.11 GB to 285.39 GB. We increased throughput from 620.62 tx/s to 2,817.98 tx/s (single-threaded) and 30,168.76 tx/s (scaled to 44 cores). We demonstrate the scalability of our off-the-chain execution environment for hard-fork testing, metric evaluations of smart contracts, and contract fuzzing.

1 Introduction

Ethereum is a permissionless blockchain capable of executing smart contracts. Since its inception in 2015, a total of 964 M transactions deployed in 11.6 M blocks have been processed. The number of unique Ethereum accounts soared from 20 M in 2018 to more than 131 M by January 2021. The rapid adoption has been fueled by applications in trade [34], banking [27, 37], governance, and supply chain. This trend has been amplified by a growing interest in decentralized finance¹, with a total value of 4 billion USD and a recent peak of 3.1 M contract calls in a single day [10].

¹Decentralized finance (DeFi) refers to an alternative, peer-to-peer financial infrastructure built on the blockchain.

Because smart contracts control large monetary values, blockchain designers and developers have a strong incentive in building tools that improve the correctness, efficiency, and security of their smart contracts. Developers typically write smart contracts in the Solidity language [17]. The Solidity compiler translates smart contracts into an immutable bytecode representation for the Ethereum virtual machine (EVM). Bytecode is persistently deployed on the blockchain where it can be invoked for execution on the EVM in a transaction.

The execution of smart contracts and Ethereum itself is a protocol defined formally in the Yellow paper [44] and practically implemented in Ethereum clients. Ethereum clients function as peers on the Ethereum network. They keep the ledger of the Ethereum blockchain consistent, fetch its updates, and integrate the EVM to execute smart contracts. The two most popular clients are Geth [18]² and Parity [39]³. When a client executes a smart contract, it constructs its input state from the blockchain's ledger and writes its state changes back onto the ledger. Hence, a smart contract's state evolves continuously on the ledger, where the history of all state changes of a smart contract is kept.

For testing, the state of a smart contract must be retrieved before it can be validated and analyzed. However, due to the ledger's cryptographically secured data structures and due to its sequential representation of the ledger as a blockchain, the retrieval of a smart contract's state is prohibitively slow [35, 46]. This problem is exacerbated for testing and debugging tools that process large quantities of blocks (or even all blocks) on the blockchain.

It has been acknowledged by the community that the testing of smart contracts at scale is a long-standing problem. It affects the development of the entire blockchain-oriented software (BOS) ecosystem, i.e., all software systems that work with a blockchain implementation. There have been calls for research into mocking blockchains [40] for enhancing testing

²Geth is the most widespread open source Ethereum client officially maintained by the Ethereum community, with a market share of 80%, according to <https://ethernodes.org/>.

³In Jan. 2020, Parity has been renamed to OpenEthereum.

and debugging in isolation. A recent survey [50] conducted among practitioners, GitHub developers, and industry professionals working on Ethereum smart contracts revealed the concern that the “*EVM is a single-threaded machine that cannot run transactions in parallel*”, which may affect “*people who have a higher requirement on the timely reaction and verification of their transactions*”. A survey [5] among 156 developers of popular blockchain projects on GitHub shows that backward compatibility (the ability to validate earlier transactions) and the difficulty of setting up testnets⁴ are pressing issues for BOS developers, and that the distributed environment currently cannot be adequately simulated on development machines.

To mitigate the above problems and enable testing and profiling of smart contracts at scale, we introduce a *transaction record and replay* mechanism that enables the fast re-execution of individual transactions for testing purposes. Our mechanism records the historical Ethereum state that was used when a transaction was originally executed. The transaction can then be replayed in isolation for testing purposes. Our recorder reorganizes the Ethereum world state into transaction-relevant substates. Replay is conducted off-the-chain to avoid the overhead of the distributed system. The replayer mocks the distributed blockchain environment so that a transaction has the same execution behavior as if it was executed by an Ethereum client. Our approach facilitates the efficient replay of all transactions of the blockchain because transactions can be replayed in parallel (unlike testnets) and in isolation. We demonstrate that our novel *off-the-chain* execution environment enables applications such as hard-fork testing (i.e., a regression test checking whether the newly introduced policies of a hard fork do not hamper the execution of legacy smart contracts), metric evaluations of smart contracts (e.g., measuring the number of wasteful instructions in a smart contract), and contract fuzzing for detecting execution anomalies in smart contracts.

The contributions of this work are as follows:

- We present a new off-the-chain execution environment for blockchain transactions using a recorder and replayer so that transactions can run in isolation showing unprecedented performance improvements in comparison with state-of-the-art approaches.
- We introduce a new substate representation for transactions that captures the substate of the world state which is relevant for their execution.
- We show the efficiency and effectiveness of our approach using a regression tester for hard forks, a dead-code analysis, and a program fuzzer.

⁴A testnet is a blockchain that is used for testing of smart contracts in a production-like environment, but without spending cryptocurrency of real value.

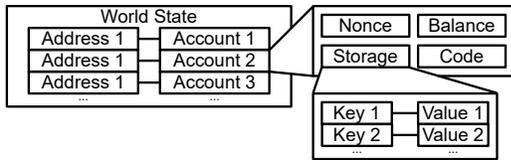
2 Background and Motivation

Ethereum Blockchain: A blockchain is a ledger shared among peers on a peer-to-peer (P2P) network. The Ethereum blockchain is fully described in its specification, the Yellow paper [44]. A *blockchain* is implemented as a chain of blocks that has been accepted by participants in the network following a consensus protocol. A *block* consists of a block header and a block body. The block header contains the reference to the parent block and metadata for the block verification. The block body holds a sequence of transactions created by participants and added in the block by miners. Transactions may either transfer funds or invoke smart contracts.

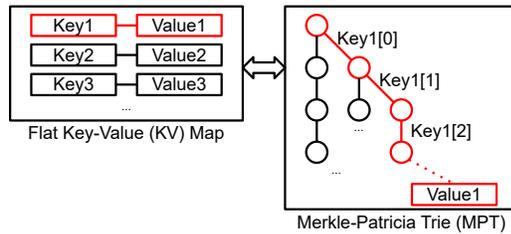
A *transaction* in the blockchain is signed by its sender and contains input data that are propagated into the global state. The execution of a transaction can be perceived as a state transition where a global state is transformed. The global state encompasses the state of smart contracts, accounts, and other aspects such as transaction receipts or smart-contract execution logs. Accounts are addressed by public keys generated by the private keys of the account owners. The sender issuing a transaction has to sign the transaction with the account’s private key. Peers can verify the transaction by computing the sender’s account address from the signatures. It is important to note that the state transition must be deterministic among all peers in the blockchain network for the sake of consistency. Without deterministic state transitions of a transaction, it is impossible to find consensus.

The Ethereum blockchain provides the following three types of transactions. A *transfer* transaction takes assets from a sender and transfers them to a recipient. A transaction for *contract creation* conveys the initial endowment of the newly created account and a smart contract in the form of EVM bytecode to initialize the created account. The created account is associated with contract code, and it has an account storage to maintain its state across contract invocations. A *contract invocation* transaction sends a message to an account to execute the associated contract code. The message contains input data and gas converted from cryptocurrency to fuel the EVM bytecode execution. The EVM will execute the contract code with the input data, consuming the gas metered based on executed instructions. If the gas consumption reaches the provided gas limit, the contract is terminated prematurely.

Ethereum World State: The Ethereum world state is the global state information of Ethereum that maps between account addresses and the data associated with an account. Each account has its own key-value storage in which only the owner has the authority to load and store values via `SLOAD` and `SSTORE` instructions of EVM bytecode. For state verification and as input to the next block, Ethereum clients are required to maintain a local copy of the entire global state after processing a particular block. As depicted in Figure 1a, Ethereum employs key-value (KV) maps to maintain (1) accounts and



(a) Ethereum world state and account storage.



(b) Flat KV map and corresponding MPT.

Figure 1: Ethereum world state and account storage (a), corresponding MPT as the underlying representation of a flat KV map (b).

(2) the data stored in each account. Data is encoded in recursive length prefix (RLP) format [44]. For cryptographic verification, the flat KV maps are encoded as Merkle Patricia tries (MPTs, [16]), depicted in Figure 1b. State verification in Ethereum relies on the property of MPTs that if two MPTs are *identical*, the hash values of their root nodes are identical [33].

The EVM bytecode interface provides the view of the flat KV map from Figure 1a to the application. Only after transaction execution will a client employ the RLP and MPT encoding depicted in Figure 1b, verify states, and store the MPT nodes in an on-disk key-value database (KVDB). A client may use a data representation different from MPTs to improve database performance [1]. Nevertheless, MPTs must still be constructed for state verification as mandated by the ledger protocol. The specification of the EVM and the world state are provided in the Ethereum Yellow paper [44].

3 Limitations of Smart Contract Testing

Testing smart contracts requires an Ethereum world state and a transaction execution environment. We refer to *transaction replay* as the execution of a transaction at the historical Ethereum state that the transaction was originally executed on at a given block height on the blockchain. Note that, in principle, it is not necessary to have the Ethereum network and blockchain available for testing. In this section, we consider different test environments currently available to blockchain designers and developers for replaying transactions. None of them facilitate replay on transaction granularity—only block granularity is supported. We highlight the impracticality of the test environments and conclude with the observed scalability problems. Unless otherwise specified, our evaluations were conducted on the server platform shown in Table 1.

Server	Geth full node	Substate replayer	Geth archive node
CPU	Intel Xeon E5-2699 v4, 2.2 GHz to 3.6 GHz, 22 cores × 2 sockets		
RAM	512 GB DDR4 RAM @ 2,400 MHz		
SSDs (PCIe)	Intel Optane 900P 480 GB Intel Optane DC P3700 800 GB	Samsung PM1725b 6.4 TB	
OS	CentOS Linux release 7.9.2009 (core), kernel version 4.11.3-1.el7.elrepo.x86_64		
Filesystem	ZFS pool (1.2 TB total size)	XFS	

Table 1: Evaluation platform specification.

3.1 Transaction Replay Delegation

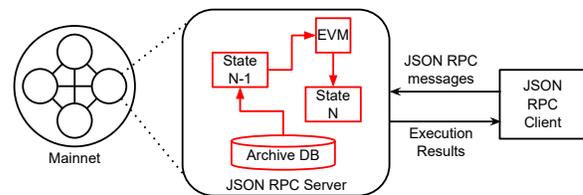


Figure 2: JSON RPC server (archive node) replaying transactions in block N on behalf of a client. State N is the state after executing all transactions in block N . The client sends request messages via the JSON RPC interface; messages include RPC method names and their arguments.

Ethereum clients can be configured as Ethereum *archive nodes*. They keep the entire history of world states in their database and can thus be used to retrieve historical state information. Geth provides a JSON RPC API⁵ for this purpose. Via the JSON RPC API, a Geth client can delegate the replay of a transaction to an archive node. Figure 2 illustrates this scenario.

Unfortunately, delegation suffers from significant overhead incurred by the JSON RPC API and the fact that transaction throughput of archive nodes is seriously constrained by the considerable database size for storing all historical world states (6 TB of disk space for blocks up to 11 M [20]). An Ethereum client can be configured as an archive node that generates execution traces during transaction replay. In a small experiment, we observed that a query to replay traces of ten blocks at blocks after 9 M using `traceBlockByNumber` timed out running for an entire hour (at that stage we stopped the replay attempt). After disabling the EVM stack, memory, and storage tracing, the archive node achieved a replay throughput of 19.4 tx/s for 100 blocks at block 9 M and after. Hence, a test environment that uses transaction replay delegation is not a viable solution for testing—even for a small number of replay queries.

⁵<https://geth.ethereum.org/docs/rpc/ns-debug>

3.2 Transaction Replay in a Client

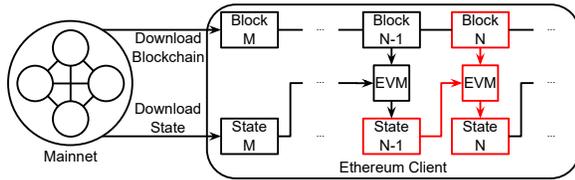


Figure 3: Full node to replay transactions in block N . State N is the state after block N has been imported, and block M is any block prior to block N on the blockchain.

When Ethereum clients are configured as Ethereum *full nodes*, they replay transactions for verification and synchronization with peers on the network. Figure 3 illustrates how Ethereum clients replay transactions during synchronization. To replay transactions in block N , a client requires the state at block $N - 1$. A full node can download the state after block M directly from trusted peers using a fast synchronization protocol instead of processing all M blocks from the genesis block by itself. The fast synchronization protocol is a trade-off to save transaction processing time at the cost of the higher network bandwidth required for downloading the state. The community refers to fast synchronization as *fast-sync* mode and it represents the default configuration of Geth clients. At block 11.5 M, a Geth full node consumes more than 600 GB of disk space [21].

Transaction replay in full nodes suffers from severe drawbacks: If the desired block for replay is located far behind the tip of the chain, no peer will be available to provide the state for synchronizing in fast-sync mode. In the absence of a downloadable state, the client must process (import) all blocks itself, commencing from the genesis block, to produce state $N - 1$ for replaying block N . In our experiment, the Geth client on an AWS `i3.2xlarge` instance took 8.13 h to fast-sync the state at block 11.5 M. When the Geth client imported blocks from the genesis block, it took 227 h (i.e., 9.5 d) to produce the state at block 8 M. Similarly to replay delegation, transaction replay throughput is very low: At block height 9 M the observed throughput of our archive node was 19.4 tx/s while our full node achieved 447.7 tx/s. Therefore, replaying millions of transactions with an archive node is not viable in practice, and a full node will take several weeks to accomplish such a testing task.

3.3 Testnets

The Ethereum main network (mainnet) is the only P2P network for real-world trading of Ethereum cryptocurrency. For testing purposes, the Ethereum community maintains public and private testnets where new protocols are activated and tested before they are deployed on the mainnet. Blockchain designers and developers deploy and test contracts on a test-

net using clients that synchronize with recent blocks from the testnet like the nodes on the mainnet. The major limitation of testing with testnets is that its state can only be configured via side effects of smart contracts that are executed through transactions. This requires considerable effort from a developer to build a test fixture. In particular, the developer must deploy and execute the following transactions to test a smart contract on a testnet: (1) transactions to initialize the target contract and other participating accounts, (2) transactions to set up the complete world state and environment parameters for the input, and (3) a transaction to invoke the target contract with the prepared input state.

3.4 Lack of Scalability on Multicores

Testing smart contracts by replaying transactions on a full node does not scale on a multicore server because of two reasons: First, execution of transactions on the blockchain is inherently sequential. To replay transactions in blocks later than the current state, a node must import all intermediate blocks on the chain between the current block and the block to be replayed. Second, it takes a considerable amount of disk space to maintain a complete world state in MPT format. Running multiple Ethereum nodes with different ranges of blocks has the potential to increase the overall throughput on a multicore server. However, each node will require hundreds of GBs to maintain multiple world state instances. E.g., running nine full nodes for segments of 1 M blocks as stated in Table 2 requires 2.8 TB of disk space. One may expect an archive node to scale on multicores because it does not have to import blocks prior to replaying a transaction. But practically, as observed in our experiments, a single archive node requires 6 TB of disk space and exhibits very low transaction replay throughput. (Note that our evaluation platform from Table 1 uses PCIe SSDs and thus can be considered an advantageous case, performance-wise.) Testing smart contracts on a testnet has the same limitations as replaying of transactions on a full node because it executes transactions in order and has to encode a complete state in MPT format for verification.

4 Off-The-Chain Testing

Our new *off-the-chain* testing environment is based on a transaction record and replay mechanism. The record mechanism records the historical state from before and after a transaction on the chain. The recorder is an augmented Ethereum client that produces the historical state as a side effect while importing blocks from the blockchain. For a transaction, the recorder captures *substates*, which contain only relevant parts of the world-state and environment parameters that contain enough information to replay a transaction with no dependency on earlier transactions. The recorder stores the substates in the *substate database*, which is a flat key-value map.

The replay mechanism mocks the Ethereum protocol of a client such that transactions can be executed without the peer-to-peer network of Ethereum and the cryptographic verification tasks of a client. We use EVM binaries for building the replay mechanism, which are stand-alone clients and have been implemented for checking the correctness of the EVM only. Our replay mechanism operates directly on the substate database. It executes transactions off-the-chain, in parallel, and in isolation achieving unprecedented scalability for testing smart contracts.

4.1 Transaction Substate

Key to our record/replay mechanism is the substate, which contains a subset of the world-state trie to faithfully replay a transaction. The subset contains all the entries represented as a key-value pair (not as an MPT) for executing the transaction in full isolation. The substate is enhanced with meta information that is required for the execution of the transaction including (1) the values of the input arguments of the transaction, (2) the cryptographic hashes needed for the execution, and (3) the expected return values. In the following, we summarize the information stored in substates.

1. **Alloc:** information about each accessed account, i.e., account address, nonce, balance, code hash, and storage values from the world state accessed by the transaction.
2. **Block:** block number, block creation timestamp, block hashes, coinbase, difficulty, and block gas limit.
3. **Message:** nonce, gas price, provided gas, sender account address, recipient account address, input value, and input data of the transaction to initiate the message call.
4. **Result:** status code, transaction gas usage, logs and their Bloom filter from the output of the transaction execution.

The substate representing the historical Ethereum world state before executing a transaction is called the *input substate*. It consists of three parts: input alloc, block, and message. The substate representing the historical Ethereum world state after executing a transaction is called the *output substate*. It consists of two parts: output alloc and the transaction result. Together, the input and output substates contain all information required to faithfully replay and validate the respective transaction. Note that accounts and storage values that are not accessed by a transaction can be omitted from a substate because they have no effect on the transaction’s execution and hence cannot affect the transaction’s output.

4.2 Substate Recorder

Our substate recorder is an extended Ethereum client that collects and stores transactions’ substates while importing blocks as depicted in Figure 4. Our recorder collects the substates from before and after a transaction on the chain. We refer

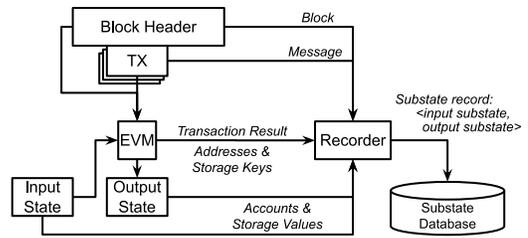


Figure 4: Substate recorder.

to these substates as the input and output substates, respectively, and the recorder stores them as a substate record in the substate database.

Throughout the execution of a transaction, the recorder collects the set of *indices* accessed in the world state. An index is represented by a tuple $\langle \text{Address}, \text{Key} \rangle$, where *Address* denotes an account/wallet address, and *Key* represents the storage key to the account’s accessed storage location (as illustrated in Figure 1a). After the transaction’s completion, the recorder collects for each index the corresponding value from the world state prior to the start of the transaction (for the input substate), and the value from the world state after the transaction terminated (for the output substate). The collected *input* and *output tuples* are of the shape $\langle \langle \text{Address}, \text{Key} \rangle, \text{Value} \rangle$. Collectively, the input/output tuples contain the complete set of storage locations and associated values read and written by a transaction.

As an example, we assume a *token transfer* of 10 units from account 1 at *address1* to account 2 at *address2*. Account 1 and account 2 are endowed with 25 units and 80 units before transaction commencement, and both accounts maintain their token values at storage key *key2*. Recording this transaction will result in the following input/output tuples, which signify the drop of funds in account 1 from 25 to 15 units, and the increase in account 2 from 80 to 90 units.

Input tuples:	Output tuples:
(1) $\langle \langle \text{address1}, \text{key2} \rangle, 25 \rangle$	$\langle \langle \text{address1}, \text{key2} \rangle, 15 \rangle$
$\langle \langle \text{address2}, \text{key2} \rangle, 80 \rangle$	$\langle \langle \text{address2}, \text{key2} \rangle, 90 \rangle$

The recorded input/output tuples do not include storage locations where accounts keep information that has not been accessed as part of a given transaction. In practice, a transaction involves only a few accounts. The recorded input/output tuples thus constitute a small subset of the world state’s account storage before and after a transaction, which facilitates space-efficient transaction replay.

We record input/output tuples for nested transactions, which occur with calls across smart contracts. Tuple collection includes reverted transactions, which are transactions that do not take effect on the world state, e.g., because the transaction runs out of gas. Even such invalid transaction termination requires the input/output tuples to ensure the faithful replay of the reverted transaction.

Consider as an example the aforementioned token-transfer contract. The contract has a runtime check for reverting a transaction if there are insufficient funds available. Assume that the initial endowment of account 1 is zero, hence, the runtime check will fail and make the transaction revert. In this case, the input substate will have the tuple $\langle\langle\text{address1}, \text{key2}\rangle, 0\rangle$ only. This tuple is nevertheless necessary for reproducing the failed runtime check during replay.

As depicted in Figure 4, our recorder creates a *substate record* from the input/output substates (including the transaction’s environment parameters, i.e., block, message, and transaction result), and stores the substate record in the substate database. Records in the substate database are indexed by the key $\langle\text{block}, \text{tx}\rangle$, where *block* is the block number and *tx* is the index of the transaction within the block. Each substate record can thereby be accessed by a single database lookup, which eliminates dependencies on earlier transactions and provides our replayer with instant access to the historical state of any recorded transaction.

Because our recorder collects substates during on-chain operations, it requires as much time and space as the synchronization of a Geth full node. However, the recording has to be performed only once. As soon as the substate database is created, all historical transactions can be replayed an arbitrary number of times with our replay mechanism.

4.3 Substate Replayer

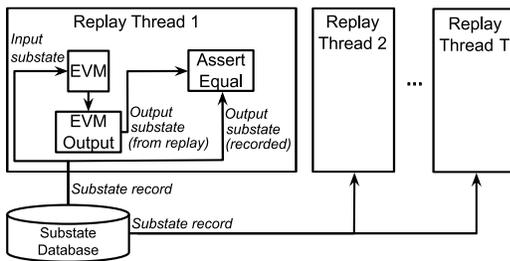


Figure 5: Substate replayer.

The replay mechanism is based on the EVM binaries that we extended for replaying transactions arbitrarily and in parallel from our substate database. Figure 5 illustrates the design of our multi-threaded replayer. It receives the number of replay threads T and an inclusive block range $[N_{\text{first}}, N_{\text{last}}]$ as input. The replayer creates a thread pool of T replay threads and pushes the designated block numbers from N_{first} to N_{last} into a channel on which all replay threads wait⁶. We employ a dynamic work partitioning scheme where replay threads repeatedly dequeue a block number, load the block’s substate records, and replay its transactions.

⁶The EVM binaries and hence our extensions are implemented in the Go programming language.

For each transaction, a replay thread invokes a fresh EVM instance, for which it copies input alloc, block, and message from the input substate to an in-memory context that the EVM instance can read and modify off-the-chain. This in-memory context simulates on-chain features such as hard forks, gas costs, block environment, precompiled contracts, and database snapshots for reverted transactions.

The EVM instance executes the corresponding smart contract code in isolation on the in-memory context. The replay thread thereafter collects the output tuples and the transaction result from the modified context and validates them against the recorded output substate. If a thread successfully replayed all transactions in a block, it sends the block number back to the main thread. The main thread maintains the processed block total and triggers the termination of the replay threads once all blocks have been processed.

Replay threads collect the set of indices accessed in the world state in the same manner as the recorder. Because transaction replay must be deterministic (running the same transaction with the same input must produce the same output), the replayed transaction must access the same storage locations as the recorder, and the accessed indices must coincide.

For example, considering the token transfer example (1) on the previous page. If during replay the transaction accesses index $\langle\text{address1}, \text{key1}\rangle$ instead of $\langle\text{address1}, \text{key2}\rangle$, the replayer will detect this index mismatch (cf. “Assert Equal” in Figure 5).

In general, if any of the accessed indices, output values, or the transaction result differ from the recorded historical information, the replayer will raise an exception, report the substate record’s key and the difference between the EVM output and the expected output, and terminate.

4.3.1 Replay Performance and Accuracy

We evaluated the runtime and storage improvements of our replayer compared to a Geth full node, and we validated its replay accuracy. All experiments were conducted on the platform specified in Table 1. Tables 2 and 3 compare the disk space and execution-time requirements of a Geth full node and our substate replayer to replay all 590 million transactions of the initial 9 M blocks of the Ethereum mainnet. We imported blocks from files to replay transactions with the Geth full node. The size of Geth’s database in Table 2 increases drastically as it contains all accounts existing in the previous blocks. The substate database requires less space than Geth because Geth must maintain a complete world state with all accounts and storage values in MPTs for on-chain synchronization, while our off-the-chain test environment selectively recorded accounts that are accessed during transaction execution.

Table 3 compares time and throughput of transaction replay between Geth and our substate replayer with a single thread. Both took more time in later blocks because the number of transactions per block increased. Blocks in range 2–

Blocks (M)	Geth full node (GB)	Substate replayer (GB)	Space savings (%)
0-1	0.96	0.68	29.17
1-2	3.00	1.83	39.00
2-3	29.30	16.91	42.29
3-4	37.76	6.58	82.57
4-5	124.57	39.55	68.25
5-6	272.06	51.58	81.04
6-7	407.48	50.30	87.66
7-8	551.04	55.96	89.84
8-9	700.11	62.00	91.14
0-9	700.11	285.39	59.24

Table 2: Disk space requirements and space savings of our substate replayer over a Geth full node. The space required by Geth is the size of its database at the last block of the stated range. Ranges 8-9 M and 0-9 M require the same space because the Geth full node maintains a complete world state at 9 M regardless of the number of blocks it replays.

Blocks (M)	Geth full node		Substate replayer		Speed-up (×)
	Time (s)	tx/s	Time (s)	tx/s	
0-1	1184	1414.07	526	3183.01	2.25
1-2	2879	2217.13	1517	4207.73	1.90
2-3	28906	252.73	24125	302.82	1.20
3-4	10775	1946.33	5222	4016.03	2.06
4-5	94868	1196.67	28873	3931.90	3.29
5-6	165673	748.71	35390	3504.97	4.68
6-7	173503	552.82	33672	2848.55	5.15
7-8	224426	485.51	38060	2862.87	5.90
8-9	248519	447.70	41999	2649.17	5.92
0-9	950733	620.62	209384	2817.98	4.54

Table 3: Total execution time (s) and throughput in transactions per second (tx/s) of Geth block import and substate replay with a single thread.

3 M showed lower performance because of denial-of-service (DoS) attacks described in [3]. In later blocks, the Geth full node was substantially slower than our substate replayer because Geth needs multiple LevelDB lookups to read MPT nodes leading to a single MPT leaf value. As the blockchain grows, the number of MPT nodes and the size of the Geth database increases, which increases both the total number of lookups and the time per lookup. In contrast, our substate recorder packages all values required to replay a transaction into a single substate record. Therefore, one LevelDB lookup is sufficient to load all data required to replay a transaction. Overall, our substate replayer is 4.54 times faster than the Geth full node. Because transaction replay from a substate record is an isolated execution on the EVM, it can be parallelized. Figure 6 shows the speedup when using multiple replay threads for the initial 9 M blocks of the Ethereum mainnet.

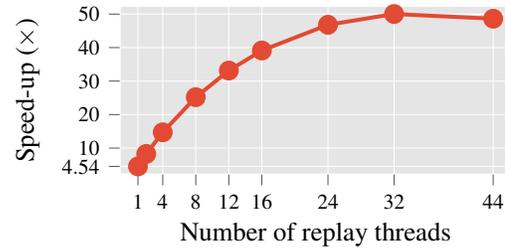


Figure 6: Speedup of parallel transaction replay by the multi-threaded substate replayer with 9 M blocks compared to a Geth full node.

As discussed in Section 4.3, our replayer will raise an exception if the execution of a replayed transaction deviates from the recorded historical information. When replaying the 590 million transactions on the Ethereum mainnet from the genesis block up to block 9 M, our replayer finished without raising an exception, which confirms that all transactions were replayed faithfully and that our off-the-chain testing environment achieved a replay accuracy [26] of 100%.

5 Use Cases

In this section we introduce three use cases for our substate replayer: (1) a metric use case based on transaction replay, (2) a fuzzer use case based on testnets, (3) a method to assess hard forks using off-the-chain execution.

5.1 Metric Use Case

Program analysis techniques have been widely adopted for smart contracts. A 2019 survey of 27 Ethereum contract analysis tools [11] found that most tools concentrate on security issues and employ static analyses. But pressing efficiency and scalability limits of blockchains have become another major concern [13, 30, 41, 49]. We argue that because of the inherent limitations of static analyses, in particular precision and cost [7, 14], dynamic analysis techniques will gain further momentum. Specific metrics aimed at measuring the complexity, communication capability, gas-usage and performance have already been instigated in [40]. A set of metrics that includes contract execution time and the time to update the Ethereum world state has been proposed in [49]. Novel program metrics in smart contracts will be required to guide future design decisions for infrastructure and language implementation, e.g., to determine the profitability of speculative parallelization [12].

To demonstrate the effectiveness of our record/replay infrastructure for comprehensive dynamic analyses of transactions on the Ethereum mainnet, we implement a metric for *wasteful instructions*. A wasteful instruction is an instruction whose side effect does not lead to a lasting side effect on the

Nr.	Opcode	Nr.	Opcode
1	PUSH 0x80	13	DUP
2	PUSH 0x40	14	REVERT
3	MSTORE	15	JUMPDEST
4	PUSH 0x00	16	POP
5	PUSH 0x01	17	PUSH 0x3797
6	SSTORE	18	DUP
7	CALLVALUE	19	PUSH 0x25
8	DUP	20	PUSH 0x00
9	ISZERO	21	CODECOPY
10	PUSH 0x15	22	PUSH 0x00
11	JUMPI	23	RETURN
12	PUSH 0x00		

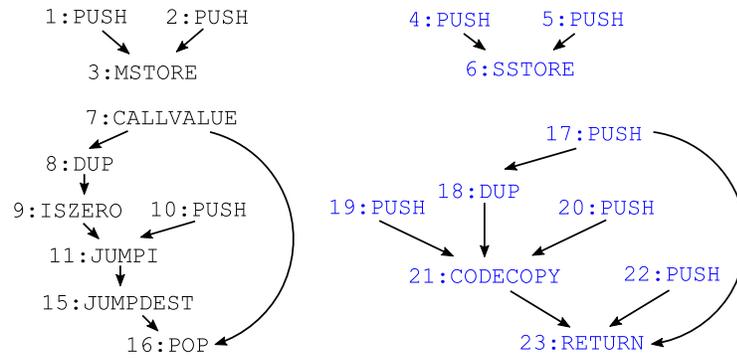


Figure 7: A bytecode of a smart contract and its value graph. Necessary instructions are colored in blue.

blockchain. Because each instruction needs to be paid for by gas, wasteful instructions are costly and should be avoided.

To determine wasteful instructions at runtime, we construct a value graph $G(V, E)$ for each smart contract execution. During the EVM execution, each instruction becomes a node in the value graph, and edges between nodes denote data flow among instructions. We define *necessary* instructions as instructions that contribute towards side effects on the blockchain or results to users. We categorize the list of necessary instructions as follows.

- SSTORE, SELFDESTRUCT, CREATE, and CREATE2 cause side effects on the blockchain by changing the world state trie or the account storage trie.
- RETURN produces a result of the smart contract and delivers it to the user.
- LOG0, LOG1, LOG2, LOG3, and LOG4 generate log records.
- CALL, CALLCODE, and DELEGATECALL execute another smart contract. Although these instructions do not have a side effect themselves, the callee contract might have a side effect. Investigating such a relationship is beyond this paper's scope, so we simplified and considered call-related instructions as necessary instructions. However, STATICCALL, which cannot have any side effect by its definition, is excluded.

A sample bytecode and its value graph are depicted in Figure 7. Each node in the graph represents an instruction with the attached number specifying the execution order. If instruction x depends on instruction y , we add a directed edge from instruction y to instruction x . For example, 3:MSTORE receives two stack values as its arguments to store the value from 1:PUSH at the memory address from 2:PUSH. Dependencies are not restricted to values from the EVM stack but may extend to memory references. Instruction 23:RETURN is such a case: it pops two arguments from the stack, which represent the address and length of data in memory to return. The third dependency (on 21:CODECOPY) encodes the memory reference itself. Instructions 12–14 from the bytecode do

not occur in the value graph because the underlying execution took the jump from instruction 11 to instruction 15 at runtime.

We built an algorithm that propagates the necessity of instructions in the value graph in a backward fashion, based on the introduced necessary instruction list. Necessary instructions are colored in blue in Figure 7. All instructions connected to 6.SSTORE and 23.RETURN are considered necessary. We obtain the set of wasteful instructions as the complement set of the necessary instructions.

To analyze the wastage characteristics of the Ethereum blockchain, we computed value graphs for the initial 9M blocks on the replayer, which took 75 h to complete. A box plot and the average ratios of wasteful instructions are shown in Figure 8 and Figure 9.

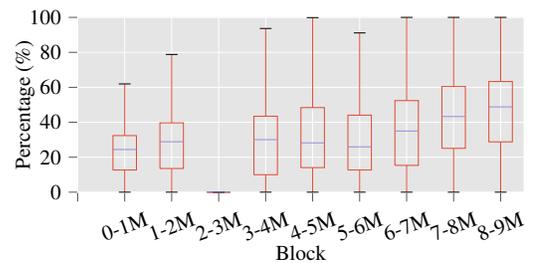


Figure 8: Box plots of wasteful instruction ratios for ranges of 1 M blocks.

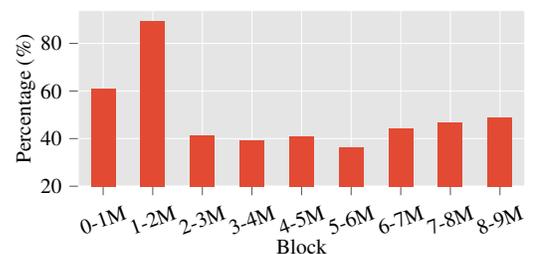


Figure 9: Average ratios of wasteful instructions for ranges of 1 M blocks.

The median and average of the wasteful instruction ratios are gradually increasing from block 5 M. As a consequence, wasteful instructions constitute nearly 50% of the total instructions in range 8–9 M. The irregularity in range 2–3 M in Figure 8 is due to DoS attacks in the year 2016 (cf. [3]). As a result, the box and whiskers in this range stick to zero, while the average ratio is not affected. Another irregularity shows in range 1–2 M in Figure 9, because it contains smart contracts with a high total number of instructions but a negligible number of necessary instructions. This skewed the average ratio of wasteful instructions, although this was not caught in the box plot.

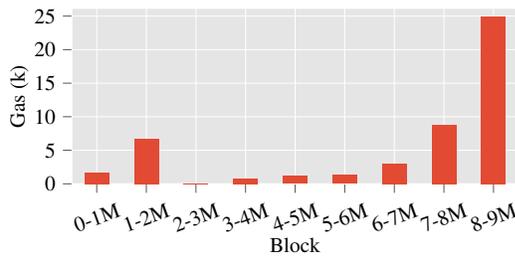


Figure 10: The average of wasted gas per transaction for ranges of 1 M blocks.

We calculated the gas consumed by wasteful instructions. As illustrated in Figure 10, recent transactions tend to waste more gas. We found a single transaction to waste nearly 25,000 gas in range 8–9 M.

5.2 Fuzzer Use Case

Fuzzing is a popular means to assess software quality [32] and exercise a program under test with many randomized inputs. A fuzzer aims to execute as many program paths as possible by varying input values, and, hence, fuzzers incur a large runtime overhead as they repeatedly execute the program under test. Some approaches reduce the environment overhead [47] or improve path coverage [36] for alleviating the fuzzing overheads. There is a variety of fuzzers available for Ethereum such as Echidna [24] and Harvey [45], which require user annotations in the Solidity source code for guiding the fuzzing process. Other fuzzing tools [31, 42] do not require annotations but still require the source code (not the EVM bytecode).

Low-performing fuzzers limit the effectiveness of a fuzzing campaign and may result in many false negatives [29]. Parallelizing fuzzers improve the performance but cannot be directly used for the blockchain testing due to its sequential execution. A parallel fuzzer for blockchains would require to replicate the blockchain for each parallel instance, which is not a viable approach.

We adapt our replay mechanism introduced in Section 4.3 for parallel fuzzing as a showcase. The aim of this showcase

is to demonstrate the efficiency of the ContractFuzzer [29] using our parallel replay mechanism. The ContractFuzzer dynamically analyzes compiled bytecode, which is most suitable for third-party auditing of binaries that does not require user annotations for fuzzing. The ContractFuzzer cannot audit third-party contracts efficiently without our replay mechanism because contracts will be fuzzed on the testnet where all smart contracts undergoing the fuzzing campaign must be deployed (as described in Section 3.3). For each input variation of a fuzzing campaign, the contracts must be redeployed on the testnet slowing down the campaign.

For fuzzing, the ContractFuzzer tool needs an Ethereum smart contract application binary interface [19, ABI], which describes input parameters and the message type of a smart contract. Most re-usable contracts/services have their ABIs publicly available, e.g., via the Etherscan web service [22].

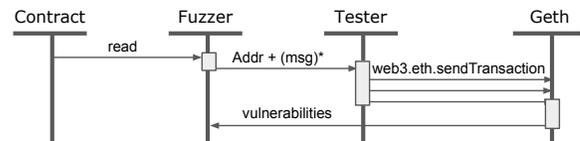


Figure 11: ContractFuzzer workflow.

Figure 11 depicts a sequence diagram of the ContractFuzzer workflow without our replay mechanism. The bytecode of a *contract* is read by the *fuzzer*, which analyzes the contract’s ABI and generates a set of random messages that execute the contract and comply with its ABI. The *tester* sends the messages one-by-one to *Geth*, for execution on the EVM, which contains specific monitoring hooks for discovering vulnerabilities that occur during bytecode execution. The results from the bytecode execution are sent back to the *fuzzer*, which generates a vulnerability report.

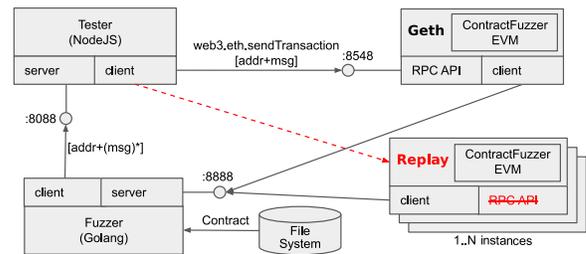


Figure 12: Extended ContractFuzzer architecture.

We extended the ContractFuzzer with our *replay* mechanism for parallel fuzzing as depicted in Figure 12. Our parallel fuzzing mechanism directly reads the input substate from the substate database (as indicated by the dashed arrow in Figure 12) such that fuzzing can be performed in parallel and without the need of deployment on the testnet.

The input variations generated by the ContractFuzzer in the form of random messages (see Figure 11) differ from the

Number of threads	Throughput (contracts/min)	Speedup (\times) over original	Speedup (\times) replay
Original	1	0.46	n/a
Replayer	1	3.28	7.07
	4	4.65	10.01
	8	6.25	13.46
	16	10.05	21.65
	32	15.99	34.44

Table 4: ContractFuzzer performance improvements.

recorded, historical *message* data of the input substate from Section 4.1. For each input variation, the replayer replaces the historical message data of the input substate by the information provided in the corresponding message from the ContractFuzzer. This modification of the transaction input obviates the need to validate the transaction (“Assert Equal” in Figure 5) because fuzzing does not attempt a faithful replay. Rather, the purpose of the input variation in the fuzzer is to execute untested program paths and uncover previously unobserved behavior in the smart contract under test. We used several of ContractFuzzer’s sample contracts [28] to validate the integration of ContractFuzzer and its modified Geth EVM with our replayer. The integration produced the expected result, i.e., the same vulnerabilities as the original ContractFuzzer.

Our focus with this use case was to determine the achievable performance from off-the-chain fuzzing. For this experiment, we used our recording of the Ethereum mainnet. We created a simple NodeJS client for the Etherscan web service [22] to obtain the ABI of contracts. We used this client to get the first several hundred contracts from the mainnet that provide their ABI. Because ContractFuzzer supports contract analysis in batches, we divided this group into batches of ten contracts each. We have executed an analysis of this dataset using various combinations of batches running in parallel. We have measured the throughput of contracts (i.e., the number of analyzed contracts/min) as shown in Table 4. The first row lists the performance of the original ContractFuzzer, while the following rows list the throughput with increasing parallelism—i.e., the number of contract batches analyzed in parallel. The middle column provides the speedup over the original ContractFuzzer (w.r.t. the first row).

It follows from the second row that the ContractFuzzer with replay executing all batches in sequence is faster than the original ContractFuzzer. We attribute this to the fact that Geth adds more overhead compared to our light-weight substate. The speedup grows proportionally in the number of threads. The last column shows the relative speedup of multi-threaded over single-threaded replay.

Our fuzzing method has three key benefits: Firstly, our replayer provides smart contracts instantly and without prior deployment on a testnet, making the fuzzing configuration

straightforward. Secondly, our database has no dependencies between substate records and enables parallel fuzzing. This allows fuzzing campaigns on smart contracts where an arbitrary number of threads can be utilized. Thirdly, the substate database always provides the same initial data, while fuzzing against the testnet requires repeated redeployment to start from the same initial state.

5.3 Hard Fork Assessment

The Ethereum specification has been updated over the years because of several reasons such as the introduction of new EVM instructions and the protection from DoS attacks. Peers willing to accept the update have participated by accepting a blockchain forked from the existing blockchain, or hard fork. A hard fork is first proposed via a meta Ethereum improvement proposal (meta EIP), which includes other EIPs. When the Ethereum community agrees on the hard fork, a certain block number is pre-determined after which the hard fork becomes effective.

At the time of writing there have been nine hard forks on the Ethereum blockchain that changed gas costs of existing instructions or introduced new instructions. Such updates of the EVM specification can cause problems on already-deployed smart contracts which—then—depend on an outdated specification. For example, Ethereum suffered from breaks of backward compatibility by EIP-1884 [15] during the Istanbul hard fork. The EIP changed the gas cost of several opcodes and it was expected that a few contracts will fail to operate. Indeed, Ethereum frameworks such as Aragon and Kyber found function calls in their smart contracts to fail with out-of-gas errors [6, 43]. Aragon expected that EIP-1884 would break 680 smart contracts in their framework [9], and had to release new smart contracts to replace them [43].

Therefore, it is essential to assess a hard fork on existing contracts before it is activated on the network. The Ethereum community maintains testnets where hard forks are activated and tested before being deployed on the mainnet. However, testnets have their own state databases which differ from the mainnet, and developers must execute transactions on a testnet to deploy contracts and predict possible impacts from a hard fork. This approach is merely a new execution of transactions and cannot reproduce the historical contexts with a new hard fork specification.

We propose the use of our replayer to assess new hard forks on already deployed smart contracts on the mainnet. Our replayer efficiently replays transactions in the same context except the protocols changed by the new hard fork. Hence, the effect of the hard fork on existing contracts can be observed, which helps decision making and hard-fork analysis.

For demonstration purposes we conducted an assessment of the historical hard forks on the Ethereum mainnet. Table 5 enumerates the history of the Ethereum hard forks at the time of writing. Out of nine hard forks, the *DAO Fork* and

Hard fork	Assessed transactions (M)	EVM runtime exception (%)			Output changed (%)	Gas usage changed (%)			Unaffected (%)
		Invalid JUMP	Invalid opcode	Reverted		Out-of-gas	Increased	Decreased	
Homestead	0.416	0.000	0.000	0.000	0.000	0.002	0.000	0.000	99.998
Tangerine Whistle	2.508	0.585	0.000	0.000	3.667	2.761	75.125	0.003	17.859
Spurious Dragon	3.055	0.530	0.000	0.000	9.407	2.549	71.182	0.001	16.331
Byzantium	26.014	0.062	0.000	0.000	2.560	0.299	8.359	0.001	88.718
Constantinople / Petersburg	193.668	0.008	0.000	0.000	0.344	0.040	1.123	0.000	98.485
Istanbul	303.300	0.064	0.198	1.009	3.804	7.884	68.494	18.547	0.000

Table 5: Hard fork assessment: percentage of affected and unaffected contract invocations and the share of each effect. Column “Assessed transactions” indicates the number of contract invocation transactions executed for the assessment. A transaction in “EVM runtime exception” was successful in the original invocation but raised an exception during the hard fork assessment. (1) “Invalid JUMP”: destination of the `JUMP` instruction was not a `JUMPEDEST` instruction. (2) “Invalid opcode”: executed instruction was the `INVALID` instruction or not defined in the hard fork. (3) “Reverted”: the `REVERT` instruction was executed. (4) “Output changed”: side effect on replay output changed, disregarding gas usage and account balances. (5) “Out-of-gas”: successful in the original invocation but raised out-of-gas exception in the hard fork assessment. (6) “Increased” & “Decreased”: produced the same output as the original invocation except gas usage and account balances. (7) “Unaffected”: produced the exactly same output as the original invocation.

Muir Glacier are not included because they do not affect the EVM specification for transaction execution. Hard fork *Constantinople / Petersburg* comprises two steps that took effect within the same block and hence were combined for this study. For the assessment of a hard fork on deployed contracts, all historic contract invocation transactions prior to the activation of the hard fork are relevant (cf. Section 2). The activation position of a hard fork on the chain thereby determines the number of transactions that need to be assessed, from the genesis block until the block where the hard fork became active. Column “Assessed transactions” in Table 5 depicts the number of contract invocation transactions that our replayer executed for each of the historical hard forks on the mainnet. Note that the *Istanbul* hard fork took place at block height 9.069 M, whereas the block range of our experiment was 0–9.0 M, but we do not regard the excluded 69 k blocks to be significant for this demonstration of our replayer. In total, our replayer executed 529 million contract invocation transactions as part of this assessment. The experiment took 15.15 h with an overall throughput of 9,694.30 tx/s.

The results, i.e., the effects of each hard fork on the contract invocation transactions prior to its activation on the mainnet, are depicted in Table 5. The columns below “EVM runtime exception” state the percentage of transactions for which the original invocation was successful but raised an exception in the hard fork assessment. `JUMP` instructions to invalid destinations and the `INVALID` instruction have been used as a pragmatic way to throw runtime exceptions. The *Byzantium* hard fork at block 4,370,000 introduced the `REVERT` instruction. The main difference is that `JUMP` and `INVALID` consume all remaining gas but `REVERT` refunds the remaining gas to the sender.

Effects on Execution Path The sum of column “EVM runtime exception” and column “Output changed” is the percentage of transactions for which the EVM specification change resulted in an execution path different from the original invocation. The hard fork with the highest ratio of sum of “EVM runtime exception” and “Output changed” is *Spurious Dragon* with 9.9 %, followed by *Istanbul*, and *Tangerine Whistle* with 5.1 % and 4.3 %. *Spurious Dragon* activated EIPs that increased the gas cost of EVM instructions, limited code size, and cleared empty accounts in the world state trie to protect the network from DoS attacks. The other two hard forks, *Istanbul*, and *Tangerine Whistle*, mainly updated gas costs of EVM instructions. This is an indication that the execution paths of those contracts are highly dependent on the EVM gas system. Hence, updating the gas system may cause such contracts to fail.

Effects on Gas Consumption The hard forks *Tangerine Whistle* and *Spurious Dragon* increased the gas costs of several EVM instructions, resulting in more than 70 % of contract invocations to consume more gas, and 2 % to raise an out-of-gas exception. The *Istanbul* hard fork increased the gas costs of several instructions that access tries and reduced the gas costs of loading call data input, which affected all contract invocations. As a result, 68.5 % of all contract invocations consumed more gas, 7.9 % raised an out-of-gas exception, and 18 % consumed less gas.

6 Related Work

Hartel and Stalduin [26] proposed a tool to generate a replay script for historic transactions based on Truffle. It redeploys the smart contract with historic data for replay on the blockchain. This approach cannot faithfully reproduce historic data because the execution environment may have changed on redeployment. Their re-execution fails to accurately replay 39% of the 1120 sampled smart contracts [26]. In contrast, we identified the complete set of environmental parameters required to faithfully replay transactions. We achieved a replay accuracy of 100% for the 590 million transactions on the Ethereum mainnet up to block 9M. Our approach replays transactions off-the-chain and thereby eliminates the networking and consensus overhead. Our substate database provides direct access to the transaction-relevant substate of any recorded transaction, which allows transaction execution in isolation and at scale. In contrast, [26] can only execute transactions sequentially from the tip of the chain. Their method relies on the availability of the verified contract source code on Etherscan, but only 2.2% of all Ethereum smart contracts deployed until Sept. 2018 have been made available as source code on Etherscan [38].

Transaction replay is often required to find vulnerabilities of smart contracts and verify the soundness of such methods. ContractFuzzer [29] and EVMFuzzer [23] adopted fuzzing techniques to spot weaknesses in smart contracts and EVMs. ContractFuzzer used 6,991 smart contracts for fuzzing and detected seven types of vulnerabilities. EVMFuzzer performed fuzzing for 253,153 contracts to expose vulnerabilities in different types of EVMs. Hartel and Schumi [25] used mutation testing to assess the quality of smart contracts. They injected smart contract specific mutations at a large scale and performed the experiment for more than 2 million transactions.

Replaying the whole blockchain can be conducted to understand the dynamic behavior of the Ethereum ecosystem. Yang et al. [48] and Baird et al. [4] measured transaction execution time for millions of blocks and discovered that time-per-gas ratios of instructions are not uniform, which potentially threatens the decentralization of the Ethereum network. Aldweesh et al. [2] observed a similar result by measuring CPU and gas usage of opcodes independently with their OpBench framework. TokenScope [8] replayed the blockchain to investigate inconsistent tokens in Ethereum. They took a trace-based approach to find inconsistencies by comparing the information about data structure, interfaces, and events. They reported 7,472 inconsistent tokens out of 57,411 tokens from 6 million examined blocks.

7 Conclusion

The Ethereum blockchain and its surrounding environment have been rapidly developed in recent years. This has led to a situation where “the need for software engineers to de-

velop specialized tools and techniques for blockchain-oriented software development” [40] has arisen. However, there is a lack of tools that scale for tasks including third-party auditing, testing, debugging, and quality assurance.

Our work proposes a new infrastructure for the lightweight execution of smart contract transactions. Our framework can exercise smart contracts at scale, multi-threaded, and in isolation. We were able to execute all available smart contracts from the Ethereum mainnet 4.54 times faster in comparison to the standard Ethereum client, Geth. Moreover, we could scale the execution effectively, e.g., on 44 cores our framework runs 50.03 times faster.

Our infrastructure is highly suitable in scenarios that require the fast and repeated execution of smart contracts. We have demonstrated the application of our testing and profiling infrastructure in three use cases: (1) for bytecode metrics, (2) for smart contract fuzzing, and (3) for hard fork compatibility assessment. Our infrastructure scales for the whole blockchain, which has not been possible with prior approaches.

Acknowledgments

We thank our shepherd, Professor Liuba Shrira, and the anonymous reviewers for their insightful comments and suggestions.

This work was supported by Fantom Foundation, by the Australian Government through the ARC Discovery Project funding scheme (DP210101984), by the National Research Foundation of Korea (NRF) funded by the Korean government (MSIT) under Grant No. 2019R1F1A1062576, and by the Next-Generation Information Computing Development Program through the NRF, funded by the Ministry of Science, ICT & Future Planning under Grant No. NRF-2015M3C4A7065522.

Availability

The source code of our record-replay infrastructure is publicly available at <https://github.com/verovm/usenix-atc21>. The repository contains a download link to the substate database (285 GB) of the initial 9M blocks of the Ethereum blockchain that has been recorded and replayed as part of this study.

References

- [1] Alexey Akhunov. Turbo Geth. <https://github.com/ledgerwatch/turbo-geth>, accessed 2020-09-22.
- [2] Amjad Aldweesh, Maher Alharby, Maryam Mehrnezhad, and Aad van Moorsel. The OpBench Ethereum opcode benchmark framework:

- Design, implementation, validation and experiments. *Performance Evaluation*, 146:102168, 2021.
- [3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In *Proceedings of the 6th International Conference on Principles of Security and Trust*, volume 10204 of *Lecture Notes in Computer Science*, pages 164–186. Springer, 2017.
- [4] Kirk Baird, Seongho Jeong, Yeonsoo Kim, Bernd Burgstaller, and Bernhard Scholz. The economics of smart contracts, [arXiv:1910.11143 \[cs.DC\]](https://arxiv.org/abs/1910.11143), 2019.
- [5] Amiangshu Bosu, Anindya Iqbal, Rifat Shahriyar, and Partha Chakraborty. Understanding the motivations, challenges and needs of Blockchain software developers: a survey. *Empirical Software Engineering*, 24(4):2636–2673, August 2019.
- [6] ChainSecurity. Istanbul hardfork EIPs — changing gas costs and more. <https://chainsecurity.com/istanbul-hardfork-eips-increasing-gas-costs-and-more/>, 2019, accessed 2021-05-10.
- [7] Ting Chen, Zihao Li, Yufei Zhang, Xiapu Luo, Ting Wang, Teng Hu, Xiuzhuo Xiao, Dong Wang, Jin Huang, and Xiaosong Zhang. A large-scale empirical study on control flow identification of smart contracts. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019.
- [8] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. TokenScope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in Ethereum. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19*, page 1503–1520, New York, NY, USA, 2019. Association for Computing Machinery.
- [9] CoinDesk. Ethereum’s Istanbul upgrade will break 680 smart contracts on Aragon. <https://www.coindesk.com/etheriums-istanbul-upgrade-will-break-680-smart-contracts-on-aragon>, 2019-10-01, accessed 2021-05-10.
- [10] CoinDesk. Soaring DeFi usage drives Ethereum contract calls to new record. <https://www.coindesk.com/soaring-defi-usage-drives-ethereum-contract-calls-to-new-record>, 2020-07-29, accessed 2021-01-07.
- [11] Monika Di Angelo and Gernot Salzer. A survey of tools for analyzing Ethereum smart contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 69–78. IEEE, 2019.
- [12] Thomas D. Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. *Distributed Comput.*, 33(3-4):209–225, 2020.
- [13] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. BLOCKBENCH: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data - SIGMOD '17*, pages 1085–1100, Chicago, Illinois, USA, 2017. ACM Press.
- [14] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 530–541, 2020.
- [15] Ethereum. EIP-1884: Repricing for trie-size-dependent opcodes. <https://eips.ethereum.org/EIPS/eip-1884>, 2019, accessed 2021-05-10.
- [16] Ethereum. Merkle Patricia Tree. <https://eth.wiki/en/fundamentals/patricia-tree>, accessed 2020-12-10.
- [17] Ethereum. Solidity language documentation. <https://docs.soliditylang.org>, accessed 2020-12-20.
- [18] Ethereum. Go Ethereum (Geth) client. <https://github.com/ethereum/go-ethereum>, accessed 2021-01-05.
- [19] Ethereum. Contract ABI specification. <https://docs.soliditylang.org/en/develop/abi-spec.html>, accessed 2021-05-07.
- [20] Etherscan. Ethereum full node sync (archive) chart. <https://etherscan.io/chartsync/chainarchive>, accessed 2021-01-11.
- [21] Etherscan. Ethereum full node sync (default) chart. <https://etherscan.io/chartsync/chaindefault>, accessed 2021-01-11.
- [22] Etherscan. Etherscan API for smart contracts’ ABIs. <https://etherscan.io/apis>, accessed 2021-05-05.
- [23] Ying Fu, Meng Ren, Fuchen Ma, Heyuan Shi, Xin Yang, Yu Jiang, Huizhong Li, and Xiang Shi. EVMFuzzer: detect EVM vulnerabilities via fuzz testing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1110–1114, 2019.

- [24] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 557–560, 2020.
- [25] Pieter Hartel and Richard Schumi. Mutation testing of smart contracts at scale. In *International Conference on Tests and Proofs*, pages 23–42. Springer, 2020.
- [26] Pieter Hartel and Mark van Staalduinen. Truffle tests for free – replaying Ethereum smart contracts for transparency, [arXiv:1907.09208 \[cs.SE\]](https://arxiv.org/abs/1907.09208), 2019.
- [27] Anna Irrera. Northern Trust uses blockchain for private equity record-keeping. <https://www.reuters.com/article/nthern-trust-ibm-blockchain/northern-trust-uses-blockchain-for-private-equity-record-keeping-idUSL1N1G61TX>, 2017, accessed 2021-01-08.
- [28] Bo Jiang. Ethereum vulnerable smart contract benchmark. <https://github.com/gongbell/ContractFuzzer/tree/master/examples>, commit 4e69adb27b42213a045623d37bfe890a20b4ff05.
- [29] Bo Jiang, Ye Liu, and WK Chan. ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 259–269. IEEE, 2018.
- [30] John Kolb, Moustafa AbdelBaky, Randy H. Katz, and David E. Culler. Core concepts, challenges, and future directions in blockchain: A centralized tutorial. *ACM Comput. Surv.*, 53(1), February 2020.
- [31] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68. IEEE, 2018.
- [32] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.
- [33] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.
- [34] Ron Miller. IBM teams with Maersk on new blockchain shipping solution. <https://techcrunch.com/2018/08/09/ibm-teams-with-maersk-on-new-blockchain-shipping-solution>, 2018, accessed 2021-01-08.
- [35] Gianmaria Del Monte, Diego Pennino, and Maurizio Pizzonia. Scaling blockchains without giving up decentralization and security: a solution to the blockchain scalability trilemma. In *Proceedings of the 3rd Workshop on Cryptocurrencies and Blockchains for Distributed Systems*, pages 71–76, 2020.
- [36] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 787–802. IEEE, 2019.
- [37] Federal Reserve Bank of Boston. Beyond theory: Getting practical with blockchain. <https://www.bostonfed.org/publications/fintech/beyond-theory-getting-practical-with-blockchain/building-an-ethereum-blockchain-proof-of-concept.aspx>, white paper, 2019, accessed 2020-12-20.
- [38] Gustavo Ansaldi Oliva, Ahmed E. Hassan, and Zhen Ming (Jack) Jiang. An exploratory study of smart contracts in the Ethereum blockchain platform. *Empir. Softw. Eng.*, 25(3):1864–1904, 2020.
- [39] OpenEthereum. Parity Ethereum client, version 2.4.0. <https://github.com/paritytech/parity-ethereum/releases/tag/v2.4.0>.
- [40] Simone Porru, Andrea Pinna, Michele Marchesi, and Roberto Tonelli. Blockchain-oriented software engineering: Challenges and new directions. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 169–171, Buenos Aires, Argentina, May 2017. IEEE.
- [41] Scott Ruoti, Ben Kaiser, Arkady Yerukhimovich, Jeremy Clark, and Robert Cunningham. Blockchain technology: What is it good for? *Commun. ACM*, 63(1):46–53, December 2019.
- [42] Noama Fatima Samreen and Manar H Alalfi. Reentrancy vulnerability identification in Ethereum smart contracts. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 22–29. IEEE, 2020.
- [43] Brett Sun. Impact of the Istanbul hard fork on Aragon organizations. <https://aragon.org/blog/istanbul-hard-fork-impact>, 2019-11-30, accessed 2021-05-10.
- [44] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, Petersburg version 3e2c089 – 2020-09-05. <https://ethereum.github.io/yellowpaper/paper.pdf>, accessed 2020-09-22.

- [45] Valentin Wüstholtz and Maria Christakis. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1398–1409, 2020.
- [46] Junfeng Xie, F Richard Yu, Tao Huang, Renchao Xie, Jiang Liu, and Yunjie Liu. A survey on the scalability of blockchain systems. *IEEE Network*, 33(5):166–173, 2019.
- [47] 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*, pages 2313–2328, 2017.
- [48] Renlord Yang, Toby Murray, Paul Rimba, and Udaya Parampalli. Empirically analyzing Ethereum’s gas mechanism. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 310–319. IEEE, 2019.
- [49] Peilin Zheng, Zibin Zheng, Xiapu Luo, Xiangping Chen, and Xuanzhe Liu. A detailed and real-time performance monitoring framework for blockchain systems. In *Proceedings of the 40th International Conference on Software Engineering Software Engineering in Practice - ICSE-SEIP '18*, pages 134–143, Gothenburg, Sweden, 2018. ACM Press.
- [50] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach D. Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. Smart Contract Development: Challenges and Opportunities. *IEEE Transactions on Software Engineering*, 2019.

Octo: INT8 Training with Loss-aware Compensation and Backward Quantization for Tiny On-device Learning

Qihua Zhou[†], Song Guo[†], Zhihao Qu[‡], Jingcai Guo[†], Zhenda Xu[†],
Jiewei Zhang[†], Tao Guo[†], Boyuan Luo[†], Jingren Zhou^{*}
[†]Hong Kong Polytechnic University, [‡]Hohai University, ^{*}Alibaba Group

Abstract

On-device learning is an emerging technique to pave the last mile of enabling edge intelligence, which eliminates the limitations of conventional in-cloud computing where dozens of computational capacities and memories are needed. A high-performance on-device learning system requires breaking the constraints of limited resources and alleviating computational overhead. In this paper, we show that employing the 8-bit fixed-point (INT8) quantization in both forward and backward passes over a deep model is a promising way to enable tiny on-device learning in practice. The key to an efficient quantization-aware training method is to exploit the hardware-level enabled acceleration while preserving the training quality in each layer. However, off-the-shelf quantization methods cannot handle the on-device learning paradigm of fixed-point processing. To overcome these challenges, we propose a novel INT8 training method, which optimizes the computation of forward and backward passes via the delicately designed *Loss-aware Compensation* (LAC) and *Parameterized Range Clipping* (PRC), respectively. Specifically, we build a new network component, the compensation layer, to automatically counteract the quantization error of tensor arithmetic. We implement our method in Octo, a lightweight cross-platform system for tiny on-device learning. Evaluation on commercial AI chips shows that Octo holds higher training efficiency over state-of-the-art quantization training methods, while achieving adequate processing speedup and memory reduction over the full-precision training.

1 Introduction

The unprecedented booming of Machine Learning (ML) techniques has achieved great success in the past decade [8, 35]. The magic of ML comes from model training on large-scale datasets, relying on the deployment in the cloud environment to meet the resource-hungry demands [44, 45, 60]. This kind of in-cloud computing paradigm can bring about the essential drawbacks that it is hard to provide personalized models

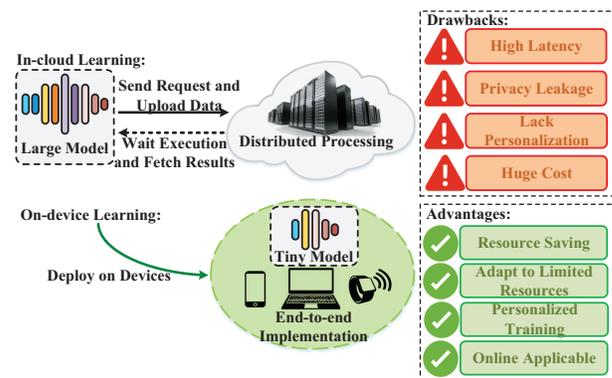


Figure 1: Conventional ML applications rely on the in-cloud learning paradigm, incurring essential drawbacks. A new trend is to use on-device learning to address these issues.

[14], suffering from high latency [46] and privacy leakage [40]. These issues promote the rise of on-device learning [5, 39, 65], which eliminates in-cloud learning’s limitations by handling the end-to-end learning process totally on user devices [4, 57], *e.g.*, mobile and IoT equipment [54]. This edge environment makes on-device learning often subject to limited computational capacity and memory [66]. Therefore, breaking resource constraints is the key step to implement on-device learning systems.

Conventional model compression methods, *e.g.*, Low-rank Decomposition [31, 61–63], Model Pruning [11, 16, 18, 20] and Network Sparsification [1, 19, 37], are insufficient as they are designed for large-scale training tasks and cannot well match the characteristic of tiny on-device learning. Fortunately, previous researches reveal that neural networks are often implemented in a 32-bit floating-point (FP32) format [17] and representing model parameters in such high precision is not always necessary [36]. It is feasible to represent parameters in lower bit precision while not downgrading the entire network quality. Based on our preliminary experiments (§2.4) that handle CNN training in the 8-bit fixed-point (INT8) format [2], we are inspired by the potential improvements and

thus intend to design a full-INT8 quantization-aware training system for on-device learning.

However, existing quantization methods can hardly be employed for on-device training due to the following limitations: (1) cannot apply to the training process [3, 5, 24, 36, 42], (2) cannot support generic networks without specific structure design [34, 47, 58], (3) cannot enable hardware-level INT8 acceleration during the training phase [7, 23, 49, 56], and (4) cannot make the gradient calibration fit on-device resource restrictions in backward pass [32, 67]. Therefore, we need a full-INT8 training method working on devices directly, covering both forward and backward passes.

Designing a desired INT8 on-device training is not easy, and we need to overcome the following challenges: (1) simplifying the computational procedure and truly accelerating processing speed on devices, (2) maintaining model quality when using INT8 quantization-aware training, (3) alleviating system overhead, *e.g.*, reducing memory footprint and I/O bandwidth utilization, and (4) making the system ease-of-use and compatible with multiple platforms. We achieve these objectives via a co-design of neural network constructor and 8-bit training engine.

More precisely, we give deep insights into the rationale of 8-bit training (§4.1) and present a theoretical analysis of layer-wise quantization error (§4.2). We point out the key to preserve model quality in forward pass is to fill the error gap in tensor dot products and thus propose the *Loss-aware Compensation* (LAC) method (§4.3) to approximately recover the quantized output to the full-precision domain. Specifically, we propose a new network component, the compensation layer (§4.3.1), to handle this adjustment with three learnable parameters, and design an L2-regularizer (§4.3.2) to bound the update rate of these parameters for convergence stability. Meanwhile, we optimize the gradient calculation in backward pass by abstracting the calculation of derivative flows (§4.4.1) as a series of primary instructions (*e.g.*, dot product, broadcasting and bit-wise shifting) and introducing symmetric quantization on them. We carefully profile the possible distribution of gradient tensors and propose the *Parameterized Range Clipping* (PRC) method (§4.4.2) to handle INT8 quantization of intermediate derivatives. We address the issue of zero point offset in backward pass by restricting clipping domain in the symmetric scheme, under 95% confidence interval of gradient distribution.

We implement our INT8 training method in Octo, a lightweight cross-platform system for on-device learning tasks. Octo’s training engine and network constructor are built in pure Python without the dependency of other sophisticated third-party libraries, and thus it is easy to port Octo to embedded devices. We evaluate Octo in different operating systems and deploy it on commercial AI platforms, such as HUAWEI Atlas 200DK [26] and NVIDIA Jetson Xavier [27], which can utilize the dedicated INT8 neuron chips. The evaluation results show that Octo achieves higher training

efficiency over state-of-the-art quantization training methods, with tiny system overhead. Specifically, Octo accelerates processing speed by up to $2.03\times$ and saves memory footprint by up to $3.37\times$, over the full-precision training.

Overall, the key contributions of our work are as follows:

- We propose a lightweight INT8 training method, enabling data quantization for both forward and backward stages (§4.1), which is efficient for deploying on-device learning applications.
- We present the theoretical characterization of quantization errors and reveal that filling the error gap caused by inner product is the key to preserve model quality of quantization-aware training (§4.2). This analysis guides us to design the compensation term to adjust quantized convolutional output.
- To achieve stable training efficiency, we propose the *Loss-aware Compensation* (LAC) and *Parameterized Range Clipping* (PRC) methods to handle data quantization in forward pass (§4.3) and backward pass (§4.4), respectively. LAC introduces the novel compensation layers and updates the compensation term based on the feedback of network loss. Meanwhile, PRC maintains bit precision in gradient calculation and avoids offset error of the zero point via symmetric clipping.
- We build a cross-platform system named Octo, which is compatible with different operating systems and can be easily ported to embedded platforms (§5). Octo is specifically designed for on-device training by exploiting the fixed-point computational primitives of embedded processors, with no need of GPUs. Evaluation on HUAWEI Atlas 200DK [26] and NVIDIA Jetson Xavier [27] shows that Octo achieves higher system performance over state-of-the-art quantization training methods, thus verifying the effectiveness of our approach (§6).

To the best of our knowledge, Octo is the first general framework to implement INT8 training on devices, optimizing both forward and backward stages. The project of Octo is open-source¹ and will constantly contribute to the further development of on-device learning techniques in practice.

2 Motivation

We start by discussing the limitations of conventional in-cloud learning (§2.1), and introducing the background of on-device learning applications (§2.2). Then, we will point out quantization-aware training (§2.3) is the key to deploy on-device learning in real-world scenarios, holding significant advantages over conventional model compression methods. We also present a case study to demonstrate the performance improvements (§2.4) by leveraging INT8-based training on

¹<https://github.com/kimihe/Octo>

devices, and analyze the limitations of existing quantization methods (§2.5).

2.1 Limitations of In-cloud Learning

Training ML models is a resource-hungry procedure, relying on a mass of learnable data and computational capacity. Thus, conventional ML applications are often deployed in the cloud environment, requiring expensive cost of machine clusters and distributed processing. However, such an in-cloud learning paradigm is vulnerable to privacy leakage as user information may be exposed to untrusted third parties. More seriously, the high latency for fetching inference results may become the bottleneck to slow down the learning performance. Also, as models are trained in a global scheme (*e.g.*, the Federated Learning [40]), it is not easy to provide customized models for different users to match personalized preferences [4].

2.2 Rise of On-device Learning

With the rapid growth of device processing capacity and memory volume, it comes the rise of on-device learning, which handles the end-to-end ML procedure on devices directly and breaks the limitations of in-cloud learning. A pertinent case in industry is the Apple Face ID [51], which enables personalized face recognition totally on user phones. Here, we briefly introduce the definition and key objectives of on-device learning.

Definition. In general, on-device learning refers to entirely transferring the model training and inference procedure on user devices, with no need of data exchanging to other machines [54]. Besides, on-device learning can apply to edge equipment (*e.g.*, IoT and mobile devices), thus it also corresponds to Edge Intelligence [66] in practice.

Objective #1: Resource Saving. The on-device hardware is often bounded by the resource-constrained environment, with limited computational capacity [30], memory volume [22, 33] and I/O bandwidth [38]. Therefore, saving the system overhead is the key to deploy on-device learning.

Objective #2: Model Quality. The learning algorithm should hold a stable convergence efficiency, with a comparable prediction accuracy and generalization ability as the in-cloud training schemes [39].

Objective #3: Personalized Training. As the training procedure is entirely based on the local data on user devices, it should provide customized models to meet user preference [14], instead of generating a global model for an ensemble of applications.

Objective #4: End-to-end Implementation. Considering the incremental user data, the trained models should keep up-to-date continuously. Thus, the entire learning process requires an end-to-end implementation on the devices [6].

Summary. These four objectives are the key metrics guiding the design of high-efficiency on-device learning systems.

2.3 Bit Precision and Data Quantization

Considering the resource-constrained environment on devices, compressing model size and alleviating computational pressure is the key to applying on-device learning in real-world scenarios. Although there are some common compression methods, *e.g.*, Low-rank Decomposition [31, 61–63], Model Pruning [11, 16, 18, 20] and Network Sparsification [1, 19, 37], they are designed for large-scale training tasks and cannot well match the pattern of tiny on-device learning. Fortunately, data quantization is a promising method to address these limitations.

Definition. The gist of quantization is to represent data via less bit precision, *e.g.*, converting a 32-bit floating-point (FP32) number to the 8-bit fixed-point (INT8) format. Here are two core operations of data quantization.

Operation #1: Number Discretization. The first operation is to map real numbers from a “continuous” domain to certain discrete values, *e.g.*, from floating-point numbers to integers. The number of discrete values is called the quantization level. A common way is to partition the original domain into several intervals and represent the numbers located in an interval by the central point [36]. The objective of number discretization is to reduce the value variety and represent the numbers by a few target points.

Operation #2: Domain Transformation. The second operation is to restrict the values from a wide representation range to a small range, *e.g.*, from 32 bits to 8 bits. The transformation procedure can be abstracted as a step function, where the “width” of the step can be uniformly or non-uniformly assigned [24]. Uniform transformation is hardware-friendly while non-uniform transformation can provide higher bit precision. Therefore, domain transformation aims at storing each real number in the fixed-point format with fewer bits.

Inspirations. Existing ML frameworks often implement the tensor arithmetic in FP32 or FP64 format to maintain high precision of numerical operations. However, previous work reveals that most neural networks are over-parameterized and representing model parameters in such high-precision format is not necessary [36]. It is feasible to maintain parameters in lower bit precision while not downgrading the entire network quality. Besides, numerical operations based on floating-point formats are much more expensive than fixed-point ones, especially for the tiny IoT equipment without floating-point processing units. Therefore, it comes to our motivation to compress model parameters from FP32 to INT8 format and handle the training procedure on devices.

Summary. Overall, we summarize the following properties of data quantization: First, it enables model in bit level, which can effectively reduce memory footprint and accelerate tensor arithmetic. This helps us implement on-device learning systems in resource-constrained cases. Second, quantization is more hardware-friendly for both generic hardware (*e.g.*, CPU/GPU) and specific chips (*e.g.*, FPGA), which can be

	Forward Pass (ms)	Backward Pass (ms)	Per-iteration Time (ms)	Parameter Memory (MB)	Model Accuracy
FP32	95.85	140.03	240.06	18.51	97.6%
INT8	54.57	67.66	126.41	9.42	95.2%
Comparison	1.86×	2.07×	1.89×	1.96×	-2.39%

Table 1: System performance using INT8 and FP32 training.

easily applied to edge intelligence applications.

2.4 Potential Gains

Employing data quantization into model training can save resource costs and accelerate processing speed. Here, we present an illustrative case study to show the potential gains. The experiment is the image classification task on a 3-layer CNN with MNIST dataset [9], running on the HUAWEI Atlas 200DK platform [26] with 10 epochs. We quantize the mini-batch input, weights and gradients of convolutional layers into INT8 format and inspect the system performance in terms of arithmetic efficiency, memory footprint and I/O bandwidth. From the comparison shown in Table 1, the INT8-based quantization-aware training can effectively alleviate system overhead while not downgrading the model quality. This raises an interesting question: can we achieve the same level of FP32 training performance with only INT8 operations for common on-device learning applications (*e.g.*, image classification)? Therefore, we are dedicated to building a lightweight INT8 training system to achieve this target.

2.5 Why not Existing Quantization Methods?

To implement on-device learning systems, existing quantization methods are insufficient due to the following limitations. **#1. Cannot apply to training process.** Most quantization methods are designed for inference only, where quantization is used in the forward pass based on a pre-trained model for accelerating the prediction speed [3, 5, 24, 36, 42]. As these methods have not addressed the issues of calculating gradients on discretized parameters and eliminating error gap after convolutional operations, they cannot be used in training process.

#2. Cannot support generic networks without specific structure design. Some methods aim at quantizing parameters with extremely low bit precision, *e.g.*, the binary [58], ternary [34] and XNOR [47] networks. However, they are specifically designed and require fundamentally modifications of network structures. Thus, they are not suitable for the training of generic networks.

#3. Cannot enable hardware-level INT8 acceleration in training phase. Google has proposed a verifying quantization method, called Fake Quantization [23], which uses INT8-based numerical information for parameter representation, while still packaging values in FP32 format for tensor arithmetic. The subsequent methods [7, 49, 56] also follow this paradigm that tensors are quantized and dequantized before

arithmetic operations. This paradigm cannot fundamentally accelerate processing speed or reduce memory footprint because it does not exploit the hardware-level power of fixed-point processing.

#4. Cannot make the gradient calibration in backward pass fit on-device resource restrictions. Recently, it is a trend to study the quantization-aware training with 8 bits. For example, Zhu *et al.* [67] proposed the unified INT8 training covering both forward and backward passes. However, current researches have not optimized the derivative calculation of model parameters and intermediate tensors during the backward pass, which still follows the fake quantization paradigm. As the backward pass often dominates the per-iteration time, it is of great potential to conduct backward quantization to further improve training efficiency.

Summary. Overall, we aim at designing an INT8 quantization method for training neural networks directly on devices.

3 Overview and Design Challenges

Enabling INT8 quantization on devices requires a co-design of neural network constructor and 8-bit training engine. We present Octo, an INT8 quantization-aware training system that addresses the following key challenges.

Challenge #1: How to fundamentally accelerate processing speed on devices? Existing quantization methods based on fake quantization cannot fully exploit the power of INT8 processing, thus the on-device training performance is still bounded by iterative tensor arithmetic. We need to simplify the computational procedure for more effective acceleration.

Our solution: We introduce the uniform 8-bit quantization into convolutional operations, affine blocks, activation functions and gradient calculation. The data quantization covers both forward and backward passes, thus the entire iteration can be accelerated. Also, the uniform range transformation is hardware-friendly and can be implemented in the layer wise.

Challenge #2: How to maintain model quality when using INT8 quantization-aware training? Introducing data quantization during model training will inevitably impact the numerical precision of parameters and incur accumulative errors cross layers. The final output will be significantly different from the full-precision training. Besides, the gradient calculation will face the same issue if we use quantization in the derivative calculation. These factors will degrade the final model accuracy and even make the training cannot converge.

Our solution: We maintain the model accuracy by adjusting the intermediate results of forward and backward passes together. In the forward pass, we propose the *Loss-aware Compensation* (LAC) method and design a new network component, called the compensation layer, to fill the error gap caused by quantized tensor arithmetic. The parameters inside the compensation layer will be optimized according to the network loss. For a higher updating efficiency, we introduce an L2-regularization term of compensation parameters

to modify the loss function. In the backward pass, we propose the *Parameterized Range Clipping* (PRC) to bound the transformation domain of quantized gradients, using a 95% confidence interval based on our theoretical analysis (§4.2).

Challenge #3: How to alleviate system overhead, especially reducing memory footprint? The training efficiency is often bounded by the limited memory volume and I/O bandwidth. This requires us to conduct memory and storage optimization in the training engine.

Our solution: We preserve all the parameters and intermediate derivatives in INT8 format. This effectively reduces the peak memory footprint and saves the storage cost for accessing parameter cache. Also, introducing LAC and PRC mentioned in challenge #2 may incur extra overhead. We transfer the compensation term from a higher-degree polynomial using FP32 tensors into an affine operation just relying on the output of convolutional layers. This can effectively restrict the computational cost of compensation and clipping.

Challenge #4: How to make the system ease-of-use and compatible with multiple platforms? In real-world cases, devices are often handled by tiny embedded systems that may not be well compatible with commodity learning frameworks (e.g., PyTorch Mobile [41] and TensorFlow Lite [15]). This requires us to build the system based on the standard environment without the dependency of other sophisticated third-party libraries.

Our solution: We first abstract core computation of convolutional and fully-connected layers into the basic operation of tensor-wise dot product. Then, we optimize the dot products by using pure Python and C++. We use the light-weight Pybind tool [43] and C++ header-based Eigen [52] to embed the hardware-level matrix instructions in Python-level training. This kind of hybrid implementation makes our system compatible with most operating systems, including embedded Linux, macOS and Windows.

Summary: The above four challenges and solutions guide the design of our system.

4 Octo Design

We first discuss the rationale of 8-bit training, and highlight the key difference between Octo and the prior fake quantization method (§4.1). Then, we present a theoretical formulation of quantization error and analyze how to preserve training quality (§4.2). We propose the *Loss-aware Compensation* (LAC) and *Parameterized Range Clipping* (PRC) methods to conquer the aforementioned challenges, in forward (§4.3) and backward pass (§4.4), respectively.

4.1 Workflow of 8-bit Training

We describe the gist of 8-bit training by discussing the prior work of fake quantization (§4.1.1), which is helpful for understanding the next steps of Octo design. For a clear explanation,

Variable	Definition
n	The number of bits used for quantization.
W_f	The full-precision FP32 weights.
X_f	The full-precision FP32 input.
W_q	The quantized INT8 weights.
X_q	The quantized INT8 input.
Y_f	The FP32 output of dot product in fake quantization.
Y_q	The INT32 output of dot product in Octo.
s_w	The scaling factor when quantizing W_f .
s_x	The scaling factor when quantizing X_f .
z_w	The offset of zero point when quantizing W_f .
z_x	The offset of zero point when quantizing X_f .
$Q(X)$	The quantization function.
$Q^{-1}(X)$	The dequantization function.
$\text{scale}(M)$	The function for calculating scaling factor of tensor M .
$\text{round}(M)$	The discretization function using stochastic rounding on tensor M .
$\text{clip}(M)$	The clipping function to bound the range of tensor M .
$\text{dot}(X, W)$	The dot product of X and W .
r_x	The error gap incurred by discretizing X_q .
r_w	The error gap incurred by discretizing W_q .
δ	The error gap between FP32 and INT8 dot product.
$\hat{\delta}$	The compensation term to approximate δ .

Table 2: Notations used in INT8 quantization-aware training.

we take a 3-layer CNN (1 CONV + 2 FCs) as the example and list all the notations in Table 2.

4.1.1 Fake Quantization Training

Compared with vanilla full-precision training (Figure 2(a)), fake quantization (Figure 2(b)) first quantizes the input and weight into INT8 format, then it dequantizes these variables back to the FP32 format before conducting tensor dot product. Note that the computational overhead of CONVs and FCs mainly comes from convolutional and affine operations, which can be unfolded as a series of dot products in low-level instructions. Therefore, we regard dot products as the key computation in forward pass. The rationale of fake quantization training can be described as following three steps:

Step #1: Quantization. This step transfers FP32 numbers to INT8 format.

$$X_q = \text{round}\left(\frac{X_f}{s_x} + z_x\right), \quad (1)$$

$$W_q = \text{round}\left(\frac{W_f}{s_w} + z_w\right), \quad (2)$$

$$s_x = \text{scale}(X, n) = \frac{\max(X_f) - \min(X_f)}{2^n - 1}, \quad (3)$$

$$z_x = \max(X_q) - \frac{\max(X_f)}{s_x}, \quad (4)$$

$$s_w = \text{scale}(W, n) = \frac{\max(W_f) - \min(W_f)}{2^n - 1}, \quad (5)$$

$$z_w = \max(W_q) - \frac{\max(W_f)}{s_w}. \quad (6)$$

Step #2: Dequantization. This step recovers the INT8 numbers to FP32 format.

$$X_f = (X_q - z_x) \cdot s_x \quad (7)$$

$$W_f = (W_q - z_w) \cdot s_w \quad (8)$$

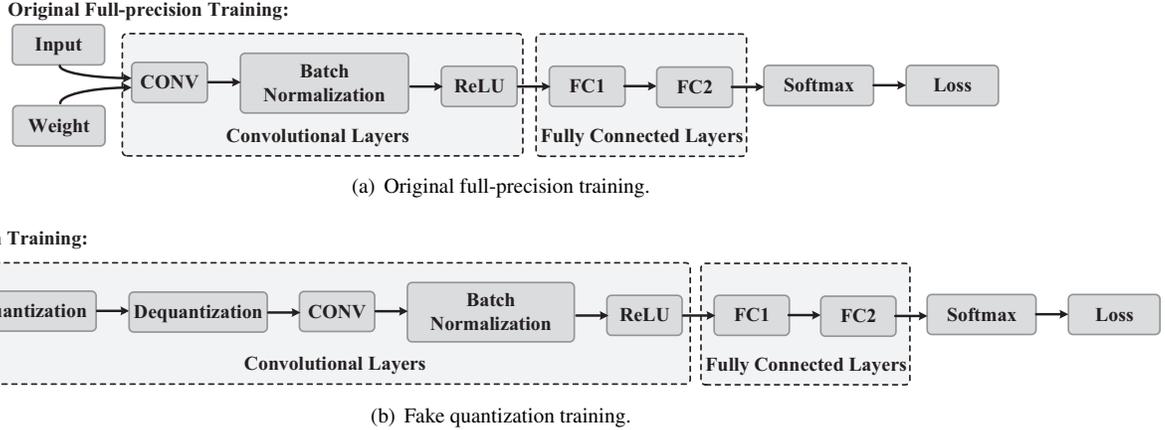


Figure 2: Workflow of previous training methods.

Step #3: Dot Products. This step conducts dot products based on the recovered FP32 tensors and yields FP32 results.

$$Y_f = \text{dot}(X_f, W_f) \quad (9)$$

Performance Analysis. Fake quantization captures the INT8 numerical information while still handling tensor arithmetic in FP32 format. Thus, *it cannot truly alleviate the computational overhead*. Also, the precision degradation after the first two steps mainly comes from the integer rounding, thus the dot products in the third step will not incur much error. However, a practical quantization-aware training method needs to move the dot products between quantization and dequantization, so as to utilize the power of fixed-point hardware. We follow this principle to design Octo’s training.

4.1.2 Octo’s Training

As shown in Figure 3, Octo holds the tensor arithmetic totally in INT8 format and contains the following three steps.

Step #1: Quantization. This step follows the same formulation as Eq. (2) and Eq. (1). We use the stochastic rounding [17] to build the discretization function $\text{round}(M)$, which maps floating-point numbers into integers and can be defined as:

$$\text{round}(M) = \begin{cases} \lfloor x \rfloor, & w.p. \quad 1 - \frac{x - \lfloor x \rfloor}{\varepsilon} \\ \lfloor x \rfloor + \varepsilon, & w.p. \quad \frac{x - \lfloor x \rfloor}{\varepsilon} \end{cases}, \quad (10)$$

where ε represents the smallest positive fixed-point value under given bits. For example, $\varepsilon = 2^{-8}$ in 8-bit quantization. This discretization function holds the unbiased rounding property, thus the expected rounding error is zero, *i.e.*, $\mathbb{E}[\text{round}(M)] = M$. This property alleviates the discretization error caused by integer rounding.

Step #2: Dot Product This step conducts dot products on INT8 tensors and returns the INT32 results to avoid overflow.

$$Y_q = \text{dot}(X_q, W_q) \quad (11)$$

Step #3: Dequantization with compensation This step converts the INT32 tensor to FP32 format and compensates the error of dot products caused by step #2.

Performance Analysis. If $z_x = 0$ and $z_w = 0$, we can simply use the following transformation to restore the INT32 output to FP32 format.

$$Y_f = Y_q \cdot (s_x \cdot s_w). \quad (12)$$

However, realistic INT8 processing requires `int` or `unsigned int` data type, which means the quantized values should be restricted within $[-128, 127]$ or $[0, 255]$, instead of using arbitrary 8-bit domains (*e.g.*, $[100, 355]$ is not feasible). Thus, we need non-zero z_x and z_w to serve as the offset of domain transformation. As z_x and z_w participate the dot product, we cannot recover Y_f by using Eq. (12), because the multiplication of zero point offset will incur a significant error gap. Compared with the vanilla dot product based on FP32 tensors, the dequantized output can be described as follows.

$$Y_f = Y_q \cdot (s_x \cdot s_w) + \delta, \quad (13)$$

where δ is the polynomial related to z_x and z_w . Therefore, step #3 is the key to Octo’s forward pass, and the gist is to *find a proper compensation term to fill the error gap δ while not incurring too much computational overhead*. The details will be discussed in the next section (§4.2).

4.2 Analysis of Error Gap

In this section, we formulate the error gap δ and approximate it as an affine transformation. Due to the page limit, we omit proof details and present the major theorem.

Theorem 1. *The error gap δ between FP32 and INT8 dot product can be approximated as an affine transformation,*

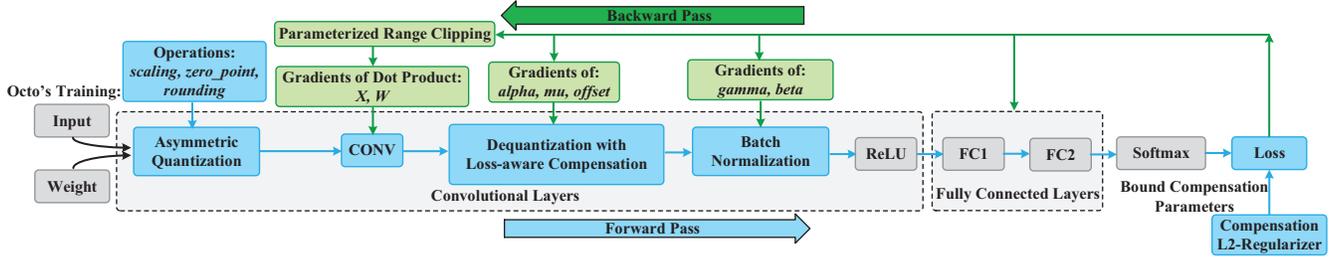


Figure 3: Workflow of Octo's training.

which is defined as:

$$\delta = s_x * \Delta X \cdot W_f + \gamma, \quad (14)$$

$$\Delta X = \frac{X_f}{s_x} - \text{round}\left(\frac{X_f}{s_x} + z_x\right), \quad (15)$$

$$\gamma = X_f(r_w - z_w)s_w - (r_x - z_x)(r_w - z_w)s_x s_w, \quad (16)$$

where $*$ represents the broadcast operation on tensors. Note that $\Delta X \cdot W_f$ dominates the computational overhead of Eq. (14), with the same shape as the dot product. Also, s_x and γ can be regarded as the coefficient factor and bias, respectively. This formulation can be simplified as an affine transformation and we approximate δ as:

$$\hat{\delta} = \alpha * \mu + \beta. \quad (17)$$

We call $\hat{\delta}$ as the compensation term, where the three parameters α , μ and β can be optimized based on the loss function of the network. Specifically, we abstract these parameters as a new network component, *i.e.*, the compensation layer (§4.3), to better adjust the quantized dot product.

4.3 Loss-aware Compensation

We propose the *Loss-aware Compensation (LAC)* method to preserve the numerical precision of tensor arithmetic in forward pass, LAC holds two core modules: (1) compensation layer (§4.3.1) and (2) L2-regularization of compensation parameters (§4.3.2).

4.3.1 Compensation Layer

The compensation layer is a new component injected to the network structure. Our target of introducing the compensation layer is to fill the error gap mentioned in Eq. (13) by approximating $\hat{\delta}$ in an affine transformation. We intend to compensate the quantized output in each layer to avoid the large accumulative errors across layers. Thus, the compensation layer is designed as an independent component that can be added at the end of both convolutional (CONV) and fully-connected (FC) layers. As shown in Fig. 4, we conduct preliminary experiments to inspect the layer completion time on CONVs and FCs. We can observe that the time cost of FCs is far less than that of CONVs. Besides, FCs usually occupy a tiny part

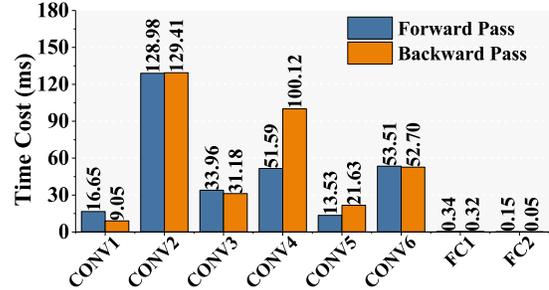


Figure 4: Layer completion time on CONV and FC layers.

of the entire network, quantizing FCs will not bring essential improvements to alleviate computational overhead. Therefore, we focus on using LAC on CONVs (although LAC can also be applied to FCs) and insert a compensation layer at the end of each CONV, following batch normalization to adjust the input distribution before ReLU, as depicted in Figure 3. Indeed, the compensation layer is an optional component, thus we can add a compensation layer after multiple CONVs. Each compensation layer holds three learnable parameters and we summarize their properties as follows.

- #1. α : This is a scalar controlling the scaling factor of compensation term.
- #2. μ : This is a tensor with the same shape as the dot product, which represents the expectation of the distribution of δ .
- #3. β : This is a tensor corresponding to the shape of μ , serving as the bias to adjust the compensation offset.

According to the expression of compensation term in Eq. (17), the calculation of $\alpha * \mu$ is handled by the broadcast operation, instead of the time-consuming tensor dot product. Also, adding β to $\alpha * \mu$ can be optimized as a shift operation, holding much less arithmetic complexity over multiplication. Therefore, calculating the compensation term will not incur much computational overhead.

4.3.2 L2-Regularization of Compensation Parameters

Although adding compensation layers can effectively fill the error gap caused by quantized dot product, the overall training efficiency may be impacted by the initialization of the compensation parameters. Actually, we can treat the approximated compensation term as special “noise” to counteract the

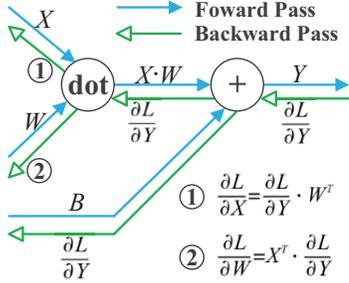


Figure 5: The derivative flows across layers in backward pass.

quantization error. Thus, the network robustness and convergence speed with compensation layers still rely on the update rule of the compensation layers. To improve the stability of compensation layers and accelerate the learning speed of compensation parameters, we design a dedicated L2-regularizer [55, 64] to better capture the significance of compensation layers. By adding the L2-regularization term to loss function, we modify the corresponding gradient calculation rule for updating the compensation parameters more efficiently. The modified loss function L is defined as follows:

$$L = -\underbrace{\frac{1}{N} \sum_{n=1}^n \sum_{k=1}^k t_{nk} \log y_{nk}}_{(1)} + \underbrace{\frac{1}{2} \lambda (\mu^2 + \beta^2)}_{(2)}, \quad (18)$$

where the first term represents the primary cross-entropy error measuring the difference between prediction y and ground truth t , based on mini-batch of size N . Meanwhile, the second term is the L2-regularizer reflecting the compensation performance from μ and β . As α is a scalar, we can omit its impact here. Actually, we can adjust the significance of these two terms by changing the value of λ . For example, we can set λ with a larger value to emphasize the effect of compensation layers. In practice, we empirically set λ as 0.1.

4.4 Backward Quantization

Apart from introducing LAC in forward pass, we also employ data quantization in backward pass as the calculation of gradients can also be abstracted as a series of dot product, following the chain rule of derivative flows (§4.4.1). However, we cannot simply use the INT8 quantization mentioned in forward pass because we cannot add a compensation-like component in backward pass to fill the error gap incurred by zero points. Here, we propose the *Parameterized Range Clipping* (PRC) method to address this issue by restricting the clipping domain in the symmetric scheme (§4.4.2).

4.4.1 Calculation of Derivative Flows

In practice, we usually calculate gradients based on chain rules instead of using numerical differential, for higher computational efficiency. As shown in Figure 5, the derivative

flows will go through the entire network, from the last layer to the first one. Capturing the flows from the $l+1$ -th layer, we can handily calculate the gradients of current l -th layer and push the flows to previous layers. Thus, the essential operations for calculating the parameter gradients ($\frac{\partial L}{\partial X}$ and $\frac{\partial L}{\partial W}$) of CONVs and FCs can also be abstracted as a series of tensor dot products, which are described as follows:

$$\frac{\partial L}{\partial X} = \text{dot}\left(\frac{\partial L}{\partial Y}, W^T\right), \quad (19)$$

$$\frac{\partial L}{\partial W} = \text{dot}\left(X^T, \frac{\partial L}{\partial Y}\right), \quad (20)$$

where W^T and X^T represent the transpose of W and X , respectively. Following these equations, we can also quantize $\frac{\partial L}{\partial Y}$ into INT8 format, and conduct the quantized dot product by using W_q and X_q , which are obtained in forward pass. After that, we can dequantize $\frac{\partial L}{\partial X}$ and $\frac{\partial L}{\partial W}$ based on the corresponding scaling factors by using Eq. (12). Note that this dequantization relies on the prerequisite of using symmetric clipping, which will be discussed in the next section.

4.4.2 Parameterized Range Clipping

As the intermediate derivative flows are also quantized before the dot product for calculating gradients, we thus need to address the issue of zero point offset. However, adding a compensation-like component in backward pass is inefficient in filling the error gap. Instead, we propose the *Parameterized Range Clipping* (PRC) method that makes clipping in the symmetric scheme to avoid the involvement of zero point, such that the FP32 parameter gradients could be recovered after conducting the INT8 dot product. The clipping range will be specifically profiled according to the distribution of the FP32 tensors. More precisely, we will discuss the clipping strategy in the following two cases.

Case #1. The distribution of FP32 tensor locates on both sides of the original point. In this case, we will quantize the FP32 tensor within $[-127, 127]$, which can be covered by the int data type. The clipping function is described as:

$$\text{clip}(M) \in [-a, a], \quad (21)$$

$$a = \min\{|\min(M)|, \max(M)\}. \quad (22)$$

Case #2. The distribution of FP32 tensor locates on one side of the original point. In this case, we will quantize the FP32 tensor within $[0, 255]$ that can be covered by the unsigned int data type. The clipping function is defined as:

$$\text{clip}(M) \in [0, a], \quad (23)$$

$$a = \max\{|\min(M)|, |\max(M)|\}. \quad (24)$$

By profiling the clipping range in these two cases, we can assert the zero point offset as 0 and surpass the issue of error gap. Moreover, we further restrict the clipping range by applying

95% confidence interval on Eq. (21), as the gradient tensors usually follow the long-tailed but bell-shape distribution.

Gradient Recovery: Based on the above clipping, we can recover the FP32 parameter gradients ($\frac{\partial L}{\partial X_f}$ and $\frac{\partial L}{\partial W_f}$) from the intermediate INT8 quantized derivative flows (Y_q , W_q and X_q) as follows:

$$\frac{\partial L}{\partial X_f} = \text{dot}\left(\frac{\partial L}{\partial Y_q}, W_q^\top\right) \cdot (s_y s_w), \quad (25)$$

$$\frac{\partial L}{\partial W_f} = \text{dot}\left(X_q^\top, \frac{\partial L}{\partial Y_q}\right) \cdot (s_x s_y), \quad (26)$$

where s_y , s_x and s_w represent the scaling factors for quantizing Y_f , X_f and W_f , respectively.

5 Implementation

We implement Octo’s training engine and network constructor in pure Python without the dependency of other sophisticated third-party libraries, making it a cross-platform system.

5.1 Network Construction

We abstract different layers for the construction of common CNNs, including CONV layer, FC layer, Batch Normalization, ReLU, Sigmoid, Pooling, Dropout, Softmax, and the proposed compensation layer. Each layer holds uniform APIs to handle the forward and backward passes. We can easily build CNNs by assembling different layers in a proper sequence. We provide a configuration file to set layer size (e.g., neuron number, filter number, filter size and padding) and initialize model parameters. We have embedded AlexNet, VGG11 and other deep CNNs in Octo, supporting the training on MNIST, Fashion MNIST and CIFAR-10/100 datasets. Moreover, we build the auto inspection module to monitor the training performance and record the key metrics, including model accuracy, training completion time, per-iteration cost, computational overhead, memory footprint and compensation performance.

5.2 Gradient Calculation

We implement the gradient calculation based on the chain rule of derivative flows. Each layer can automatically obtain its gradients by invoking the `backward` method. Here, we take the compensation layer as an example and calculate the gradients of α , μ and β by the following Python snippets:

```
def backward(self, dout)
    self.d_alpha = np.sum(dout*self.mu, axis=0)
    self.d_mu = self.alpha * dout
    self.d_beta = dout
    return dout
```

The `backward` method sequentially accepts the derivative of $\frac{\partial L}{\partial Y}$ (denoted as `dout`) from previous layers, calculates the

gradients of three compensation parameters, stores them for model updating, and returns the latest derivative flows to the next layer.

5.3 Hardware Deployment

We mainly extract the computational operations of model training into three types: (1) tensor dot product, (2) tensor broadcast, and (3) tensor addition. We optimize these operations in low-level instructions that can exploit the power of fixed-point processing units. This requires us bridging the C++ level method calling with Python-level training engine. We use the light-weight Pybind tool [43] and C++ header-based Eigen [52] to embed the hardware-level matrix instructions in Python training. This kind of hybrid implementation makes our system compatible with most operating systems, including embedded Linux, macOS and Windows. Specifically, apart from the implementation of common PC and servers, we also deploy Octo on commercial AI devices, such as HUAWEI Atlas 200DK [26] and NVIDIA Jetson Xavier [27], which can utilize the dedicated INT8 neuron chips.

6 Evaluation

We evaluate Octo on real embedded platforms in the production environment. Our key insights are as follows:

#1. Does Octo preserve model quality and how is it compared to FP32 training? Octo achieves stable convergence efficiency in different benchmarks (§6.2) and holds comparable model accuracy as FP32 training (§6.3).

#2. Can Octo improve inference efficiency? Octo accelerates image processing speed, by up to $2.03\times$ faster than FP32-based inference (§6.4).

#3. How could Octo work? Octo can effectively fill the error gap caused by data quantization and maintain tensor distribution as FP32 does. Thus, Octo preserves model quality and makes training more stable (§6.5).

#4. What is the system overhead? Octo reduces the per-iteration time cost while introducing a tiny overhead of data quantization (§6.6.1). Meanwhile, Octo saves real-time memory footprint and decreases the peak memory usage, by up to $3.37\times$ lower than FP32 training (§6.6.2).

6.1 Methodology

Testbed Setup. Considering the on-device learning characteristics, we focus on the experimental results on embedded devices. We deploy Octo on HUAWEI Atlas 200DK [26] based on Ascend 310 AI processors [13] and NVIDIA Jetson Xavier [27] equipped with dedicated INT8 neuron chips. All these devices are operated with the Ubuntu 18.04 LTS system with GNU/Linux 4.15.0-118-generic kernel.

Benchmarks. We use image classification tasks as our benchmark, based on the training of GoogLeNet [50], AlexNet

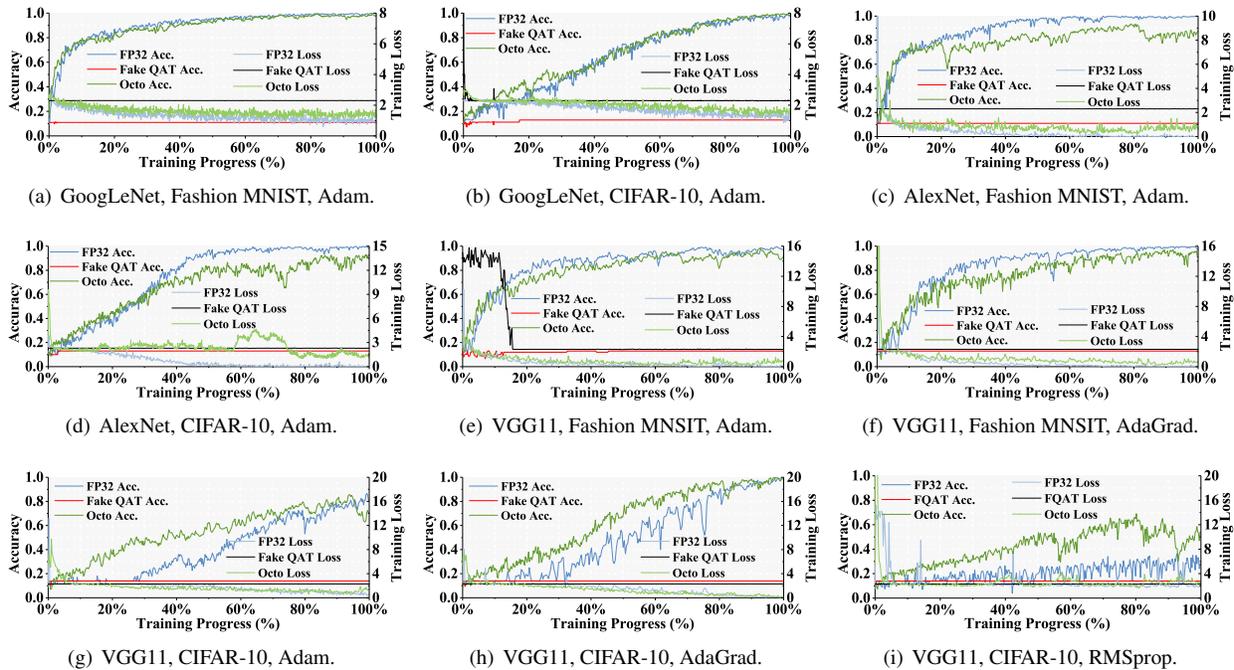


Figure 6: Training convergence efficiency using different benchmarks and optimizers.

	FP32 Acc. (%)	Octo Acc. (%)	Acc. Degradation (%)
GoogLeNet, FM	99.1 – 99.5	97.9 – 98.6	0.9 – 1.2
GoogLeNet, CF	97.8 – 99.2	97.6 – 98.8	0.2 – 0.4
AlexNet, FM	95.6 – 98.4	92.8 – 94.3	2.8 – 4.1
AlexNet, CF	91.8 – 95.2	86.1 – 87.3	5.7 – 7.9
VGG11, FM	97.5 – 98.8	94.4 – 96.5	2.3 – 3.1
VGG11, CF	97.2 – 99.5	96.5 – 98.6	0.7 – 0.9

Table 3: Comparison of FP32 and Octo’s model accuracy (from MIN to MAX) with Fashion MNIST (FM) and CIFAR-10 (CF) datasets, by using AdaGrad optimizer.

[29] and VGG11 [48]. Considering the resource constraints, large-scale datasets (*e.g.*, ImageNet [10]) are not suitable for training. We choose Fashion MNIST (FM) [59], CIFAR-10 (CF) [28] to fit the tiny on-device environment. We set the mini-batch size as 50 with 100 epochs and check Octo under Adam [25], AdaGrad [12] and RMSprop [21] optimizers.

Baselines. We build two pertinent baselines: the vanilla full-precision training (FP32) and fake quantization-aware training without error compensation (Fake QAT), using the performance metrics of training efficiency, model quality, image processing throughput and system overhead.

6.2 Training Efficiency and Quality

Maintaining convergence efficiency is one of the most crucial metrics in INT8 training design, we compare the model accuracy and training loss, by using FP32, Fake QAT and Octo training under different benchmarks in Figure 6. We can observe that Fake QAT fails to converge, where the curves of

loss (in black) and accuracy (in red) almost remain unchanged. This phenomenon indicates that putting tensor arithmetic (*e.g.*, convolutional and affine operations) between quantization and dequantization will incur huge computational errors, thus Fake QAT is not suitable for hardware-level INT8 acceleration in practice. In contrast, Octo (in green) obtains comparable accuracy as the FP32 (in blue) training, holding a just slight degradation in most cases (*e.g.*, Figure 6(a)-6(f)). Specifically, as to the training of the deep VGG11 model, Octo even achieves faster convergence speed over FP32 with close final model accuracy (*e.g.*, Figure 6(g)). Training curves using other optimizers (*e.g.*, Figure 6(h) and 6(i)) also confirm this property because the compensation layers inside Octo can serve as specific ‘noise’ to help optimizers escape saddle points. Also, the PRC method bounds the gradients in a smoother distribution, making models update more stably. Overall, the accuracy comparison and degradation gap are summarized in Table 3, where Octo preserves model quality as FP32 does and is sufficient for INT8 on-device training.

6.3 Ablation Study

We conduct the ablation study to inspect how much improvement is achieved by LAC and PRC separately. The experiments are based on the training of AlexNet model on CIFAR-10 and Fashion MNIST datasets by using Adam optimizer. Note that the LAC and PRC operations are added in CONV2, CONV3 and CONV4. We compare the average model accuracy under different training configurations in Table 4. We can observe that simply adopting INT8 data representation

	Configuration	Acc. (%)	Gap over FP32 (%)
Fashion MNIST	FP32	97.1	0
	INT8	13	-84.1
	INT8 + LAC	90.4	-6.7
	INT8 + PRC	14.8	-82.3
	INT8 + LAC + PRC	93.6	-3.5
CIFAR-10	FP32	93.5	0
	INT8	11	-82.5
	INT8 + LAC	85.2	-8.3
	INT8 + PRC	12.1	-81.4
	INT8 + LAC + PRC	86.7	-6.8

Table 4: Comparison of average model accuracy under different training configurations.

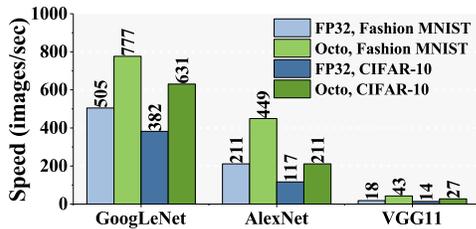


Figure 7: Image processing throughput, *i.e.*, images per second, by using FP32 and Octo trained models.

into training will significantly deteriorate the model accuracy due to large errors of the quantized dot product. Enabling LAC operation can fill this error gap by learning parameters of an affine approximation. Note that directly using PRC on vanilla INT8 training without LAC cannot eliminate the deviation of intermediate results, thus still incurring a great degradation of accuracy. However, PRC can further improve the final accuracy when LAC has been enabled, where the clipped gradients can restrict the training process in a proper convergence boundary.

6.4 Image Processing Throughput

Apart from the training efficiency, we also inspect Octo’s inference performance. We measure the image processing speed, *i.e.*, image count per second, by using the models trained by FP32 and Octo. As model parameters and tensor arithmetic are converted in INT8 format, Octo can effectively reduce I/O bandwidth and computational pressure. Therefore, Octo improves the image processing throughput, by up to $2.03\times$, on average, over FP32 model based inference. This property is significantly meaningful for on-device learning as we can reduce inference latency and improve user experience.

6.5 Octo Deep Dive

Observing the improvement of training and inference performance, we wonder how could Octo achieve this. Here, we give deep insights into how Octo compensates for quantization error and preserves model accuracy. We visualize the intermediate tensor distribution of a CONV layer’s output

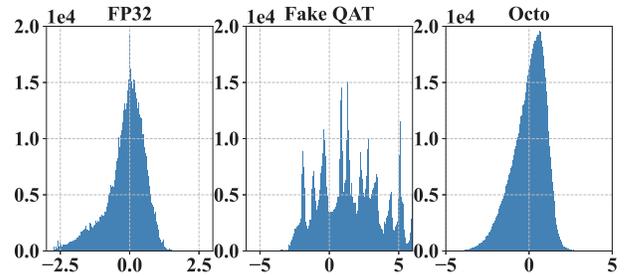


Figure 8: The visualization of tensor distribution of CONV2’s output, under FP32, Fake QAT and Octo’s training. Octo preserves similar distribution as FP32 while Fake QAT cannot.

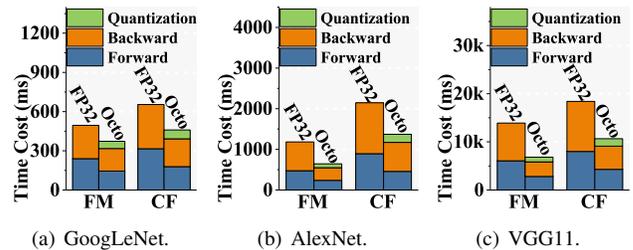


Figure 9: We inspect the details of computational time cost from 300 iterations and measure the overhead of Octo’s data quantization on average, when training Fashion MNIST (FM) and CIFAR-10 (CF) dataset.

by using FP32, Fake QAT and Octo. The data are collected from training an 8-layer deep CNN with CIFAR-10 after 80 iterations. Fake QAT holds a distinct distribution compared with FP32 because conducting dot products in the quantized domain will incur significant error to the final output. In contrast, the compensation layers inside Octo can fill the error gap and achieve similar distribution as FP32 does. Therefore, the model accuracy is maintained. Also, the PRC method in backward pass bounds derivative domains and smooths the tensor distribution, making the training more stable.

6.6 System Overhead

We compare Octo’s system overhead with FP32 training, following two key metrics of embedded platforms: computational time cost (§6.6.1) and memory footprint (§6.6.2).

6.6.1 Computational Time Cost

We measure the average computational time cost of different stages in each iteration, as depicted in Figure 9. Although Octo introduces extra overhead of data quantization, about 17.21% increase on average, it reduces the completion time of both forward and backward passes by using INT8-based processing. Overall, Octo holds shorter per-iteration time, by up to $1.73\times$ faster, on average, over vanilla FP32 training. Therefore, we believe introducing data quantization for on-device training is meaningful.

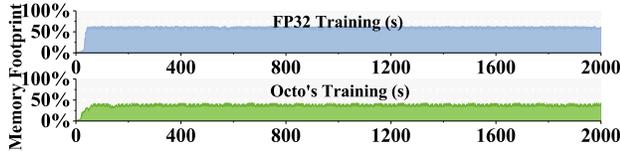


Figure 10: Octo effectively reduces real-time memory footprint over FP32 training.

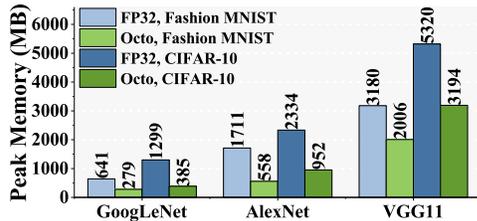


Figure 11: Peak memory usage of FP32 and Octo training.

6.6.2 Memory Footprint

We compare the memory usage under FP32 and Octo’s training, respectively. As shown in Figure 10, we monitor the real-time memory footprint when training GoogLeNet with CIFAR-10 in 2000 seconds. The FP32 training requires about 61.29% memory usage on average, while Octo only requires 40.13%. More precisely, we compare their peak memory usage of different benchmarks in Figure 11. Due to the INT8-based parameters and gradient quantization, Octo can effectively reduce peak memory usage, by up to $3.37\times$ lower than FP32 training. Such a reduction makes it possible to deploy VGG-like models.

7 Discussion

In this section, we will discuss the potential usages, extensions and limitations of Octo.

Deployment on edge devices. We mainly deploy Octo on two kinds of edge devices: (1) Atlas 200DK developer board integrating Ascend 310 AI processors and (2) NVIDIA Jetson Xavier equipped with INT8 neural chips. These devices support hardware-level INT8 operations, thus making Octo truly exploiting the power of quantization-aware training. Note that Octo is a cross-platform system and its INT8 quantization algorithm can apply to most existing ML frameworks. For example, it is possible to extend the TensorRT engine [53] to enable truly INT8 training on NVIDIA Pascal GPUs, rather than current post-quantization or inference-only INT8 usage.

Comparison with existing work. We have to build Octo from scratch due to the following limitations of existing quantization methods. PACT [7] is designed for quantized inference by optimizing the clipping range of activations in forward pass, without the consideration of gradient calculation in backward pass. Fake QAT [23] needs to pre-train a full-precision model and uses INT8 fine-tuning to preserve quantized model quality. Fake QAT only simulates INT8 cal-

culations in forward pass and cannot bring actual acceleration. Directly using Fake QAT to INT8 training will cause large errors of dot product and cannot guarantee model convergence. Unified QAT [67] enables INT8 training by adjusting gradients in backward pass. However, it relies on the calculation of both quantized and full-precision gradients, as well as plenty of exponent arithmetic. This computational overhead requires the support of GPUs and is not feasible to the device’s resource-constrained environment.

Extensions to other types of models and layers. Octo also supports other layers and models (*e.g.*, FC layers for saving more memory and RNNs for time-series prediction) because our basic optimization targets are tensor-level dot product and broadcast operations, which are prevalent in modern neural networks. Although Octo can reduce computational overhead and save memory footprint for most CNN models, we admit that Octo is just a first step to exploit the feasibility of deploying INT8 training on devices. Some complicated scenarios, such as conducting NLP on large-scale datasets or detecting real-time objects with high frame rates may still need supplementary methods, where the construction of compensation layers, clipping strategy of gradients, regularization terms of loss function should be carefully designed.

8 Conclusion

This work demonstrates that introducing INT8 quantization to training is a feasible way to implement on-device learning in practice. To truly enable hardware-level INT8 acceleration, the key of designing an efficient quantization-aware training method is to fill the error gap of dot products. This target is achieved by optimizing data quantization in both forward and backward passes, via the proposed *Loss-aware Compensation* (LAC) and *Parameterized Range Clipping* (PRC) methods, respectively. Specifically, we design a novel compensation layer to adjust the quantized output and smooth the model update procedure. Our method is implemented in Octo, a cross-platform system for tiny on-device learning. Evaluations show that Octo holds higher training efficiency over state-of-the-art quantization training methods and preserves comparable model quality as full-precision training.

Acknowledgements

This research was supported by the funding from Hong Kong RGC Research Impact Fund (RIF) with the Project No. R5060-19 and R5034-18, General Research Fund (GRF) with the Project No. 152221/19E and 15220320/20E, Collaborative Research Fund (CRF) with the Project No. C5026-18G, the National Natural Science Foundation of China (61872310), Shenzhen Science and Technology Innovation Commission (R2020A045), and Fundamental Research Funds for the Central Universities (B210202079).

References

- [1] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *ACM J. Emerg. Technol. Comput. Syst.*, 13(3):32:1–32:18, 2017.
- [2] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, pages 5151–5159, Montréal, Canada, 2018.
- [3] Ron Banner, Yury Nahshan, Elad Hoffer, and Daniel Soudry. ACIQ: analytical clipping for integer quantization of neural networks. *arXiv preprint*, abs/1810.05723, 2018.
- [4] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *Proceedings of the International Conference on Learning Representations (ICLR)*, Addis Ababa, Ethiopia, 2020.
- [5] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. Tinytl: Reduce memory, not parameters for efficient on-device learning. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
- [6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. 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 *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 578–594, Carlsbad, USA, 2018.
- [7] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. PACT: parameterized clipping activation for quantized neural networks. *arXiv preprint*, abs/1805.06085, 2018.
- [8] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel P. Kuska. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12:2493–2537, 2011.
- [9] MNIST Dataset. <http://yann.lecun.com/exdb/mnist/>, 2013.
- [10] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. Imagenet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, Miami, USA, 2009.
- [11] Xin Dong, Shangyu Chen, and Sinno Jialin Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, pages 4857–4867, Long Beach, USA, 2017.
- [12] John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, 2011.
- [13] Ascend 310 AI Processor: Energy efficiency and high integration for edges. <https://e.huawei.com/se/products/cloud-computing-dc/atlas/ascend-310>, 2021.
- [14] Biyi Fang, Xiao Zeng, and Mi Zhang. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 115–127, New Delhi, India, 2018.
- [15] TensorFlow Lite: ML for Mobile and Edge Devices. <https://www.tensorflow.org/lite>, 2020.
- [16] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, pages 1379–1387, Barcelona, Spain, 2016.
- [17] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Prithvi Narayanan. Deep learning with limited numerical precision. In *Proceedings of the International Conference on Machine Learning (ICML)*, volume 37, pages 1737–1746, Lille, France, 2015.
- [18] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding. In *Proceedings of the International Conference on Learning Representations (ICLR)*, San Juan, Puerto Rico, 2016.
- [19] Song Han, Jeff Pool, Sharan Narang, Huizi Mao, Enhao Gong, Shijian Tang, Erich Elsen, Peter Vajda, Manohar Paluri, John Tran, Bryan Catanzaro, and William J. Dally. DSD: dense-sparse-dense training for deep neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, Toulon, France, 2017.
- [20] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. *arXiv preprint*, abs/1506.02626, 2015.
- [21] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Rmsprop: Divide the gradient by a running average of its recent magnitude. In *COURSERA: Neural Networks for Machine Learning, Lecture 6.5*, 2012.

- [22] 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*, abs/1704.04861, 2017.
- [23] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2704–2713, Salt Lake City, USA, 2018.
- [24] Sangil Jung, Changyong Son, Seohyung Lee, JinWoo Son, Jae-Joon Han, Youngjun Kwak, Sung Ju Hwang, and Changkyu Choi. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4350–4359, Long Beach, USA, 2019.
- [25] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the International Conference on Learning Representations (ICLR)*, San Diego, USA, 2015.
- [26] Atlas 200DK AI Developer Kit. <https://e.huawei.com/us/products/cloud-computing-dc/atlas/atlas-200>, 2020.
- [27] Jetson AGX Xavier Developer Kit. <https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit>, 2021.
- [28] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 2012.
- [29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, pages 1106–1114, Lake Tahoe, USA, 2012.
- [30] Liangzhen Lai, Naveen Suda, and Vikas Chandra. CMSIS-NN: efficient neural network kernels for arm cortex-m cpus. *arXiv preprint*, abs/1801.06601, 2018.
- [31] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. A multilinear singular value decomposition. *SIAM J. Matrix Anal. Appl.*, 21(4):1253–1278, 2000.
- [32] Yuhang Li, Xin Dong, and Wei Wang. Additive powers-of-two quantization: An efficient non-uniform discretization for neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, Addis Ababa, Ethiopia, 2020.
- [33] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. Mccnet: Tiny deep learning on iot devices. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
- [34] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications. In *Proceedings of the International Conference on Learning Representations (ICLR)*, San Juan, Puerto Rico, 2016.
- [35] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. In *Proceedings of the European Conference on Computer Vision (ECCV)*, volume 9905, pages 21–37, Amsterdam, 2016.
- [36] Christos Louizos, Matthias Reisser, Tijmen Blankevoort, Efstratios Gavves, and Max Welling. Relaxed quantization for discretized neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, New Orleans, USA, 2019.
- [37] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J. Dally. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint*, abs/1705.08922, 2017.
- [38] Akhil Mathur, Nicholas D. Lane, Sourav Bhattacharya, Aidan Boran, Claudio Forlivesi, and Fahim Kawsar. DeepEye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 68–81, Niagara Falls, USA, 2017.
- [39] Bradley McDanel, Sai Qian Zhang, H. T. Kung, and Xin Dong. Full-stack optimization for accelerating cnns using powers-of-two weights with FPGA validation. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 449–460, Phoenix, USA, 2019.
- [40] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 54, pages 1273–1282, Fort Lauderdale, USA, 2017.
- [41] PyTorch Mobile. <https://pytorch.org/mobile/home/>, 2020.
- [42] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction. In *Proceedings*

- of the *IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1325–1334, Seoul, Korea (South), 2019.
- [43] Pybind11: Seamless operability between C++11 and Python. <https://pybind11.readthedocs.io/en/stable/index.html>, 2020.
- [44] 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 *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 307–321, 2020.
- [45] 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 ACM Symposium on Operating Systems Principles (SOSP)*, pages 16–29, Huntsville, Canada, 2019.
- [46] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: online learning of social representations. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 701–710, New York, USA, 2014.
- [47] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, volume 9908, pages 525–542, Netherlands, 2016.
- [48] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the International Conference on Learning Representations (ICLR)*, San Diego, USA, 2015.
- [49] Pierre Stock, Armand Joulin, Rémi Gribonval, Benjamin Graham, and Hervé Jégou. And the bit goes down: Revisiting the quantization of neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, Addis Ababa, Ethiopia, 2020.
- [50] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, Boston, USA, 2015.
- [51] Apple Face ID Advanced Technology. <https://support.apple.com/en-us/HT208108>, 2020.
- [52] Eigen: A C++ template library for linear algebra. <https://eigen.tuxfamily.org/index.php>, 2020.
- [53] NVIDIA TensorRT. <https://developer.nvidia.com/tensorrt>, 2021.
- [54] Mineto Tsukada, Masaaki Kondo, and Hiroki Matsutani. A neural network-based on-device learning anomaly detector for edge devices. *IEEE Trans. Computers*, 69(7):1027–1044, 2020.
- [55] Li Wan, Matthew D. Zeiler, Sixin Zhang, Yann LeCun, and Rob Fergus. Regularization of neural networks using dropconnect. In *Proceedings of the International Conference on Machine Learning (ICML)*, volume 28, pages 1058–1066, Atlanta, USA, 2013.
- [56] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. HAQ: hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8612–8620, Long Beach, USA, 2019.
- [57] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E. Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, volume 11217, pages 420–436, Munich, Germany, 2018.
- [58] Xundong Wu, Yong Wu, and Yong Zhao. Binarized neural networks on the imagenet classification task. *arXiv preprint*, abs/1604.03058, 2016.
- [59] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint*, abs/1708.07747, 2017.
- [60] 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 *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 595–610, Carlsbad, USA, 2018.
- [61] Pengtao Xie, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric P. Xing. Distributed machine learning via sufficient factor broadcasting. *arXiv preprint*, abs/1511.08486, 2015.
- [62] Pengtao Xie, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric P. Xing. Lighter-communication distributed machine learning via sufficient factor broadcasting. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, New York, USA, 2016.

- [63] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 181–193, Santa Clara, USA, 2017.
- [64] Huaqing Zhang, Jian Wang, Zhanquan Sun, Jacek M. Zurada, and Nikhil R. Pal. Feature selection for neural networks using group lasso regularization. *IEEE Trans. Knowl. Data Eng.*, 32(4):659–673, 2020.
- [65] Qihua Zhou, Zhihao Qu, Song Guo, Boyuan Luo, Jingcai Guo, Zhenda Xu, and R. Akerkar. On-device learning systems for edge intelligence: A software and hardware synergy perspective. *IEEE Internet of Things Journal*, pages 1–1, 2021.
- [66] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proc. IEEE*, 107(8):1738–1762, 2019.
- [67] Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. Towards unified INT8 training for convolutional neural network. *arXiv preprint*, abs/1912.12607, 2019.

Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism

Saar Eliad¹, Ido Hakimi¹, Alon De Jager¹, Mark Silberstein^{1,2}, Assaf Schuster¹

Technion - Israel Institute of Technology

¹*Department of Computer Science* ²*Department of Electrical Engineering*

Abstract

Fine-tuning is an increasingly common technique that leverages transfer learning to dramatically expedite the training of huge, high-quality models. Critically, fine-tuning holds the potential to make giant state-of-the-art models pre-trained on high-end super-computing-grade systems readily available for users that lack access to such costly resources. Unfortunately, this potential is still difficult to realize because the models often do not fit in the memory of a single commodity GPU, making fine-tuning a challenging problem.

We present FTPipe, a system that explores a new dimension of pipeline model parallelism, making multi-GPU execution of fine-tuning tasks for giant neural networks readily accessible on commodity hardware. A key idea is a novel approach to model partitioning and task allocation, called Mixed-pipe. Mixed-pipe partitions the model into arbitrary computational blocks rather than layers, and relaxes the model topology constraints when assigning blocks to GPUs, allowing non-adjacent blocks to be executed on the same GPU. More flexible partitioning affords a much better balance of the compute and memory-load on the GPUs compared to prior works, yet does not increase the communication overheads. Moreover, and perhaps surprisingly, when applied to asynchronous training, Mixed-pipe has negligible or no effect on the end-to-end accuracy of fine-tuning tasks despite the addition of pipeline stages.

Our extensive experiments on giant state-of-the-art NLP models (BERT-340M, GPT2-1.5B, and T5-3B) show that FTPipe achieves up to $3\times$ speedup and state-of-the-art accuracy when fine-tuning giant transformers with billions of parameters. These models require from 12GB to 59GB of GPU memory, and FTPipe executes them on 8 commodity RTX2080-Ti GPUs, each with 11GB memory and standard PCIe.

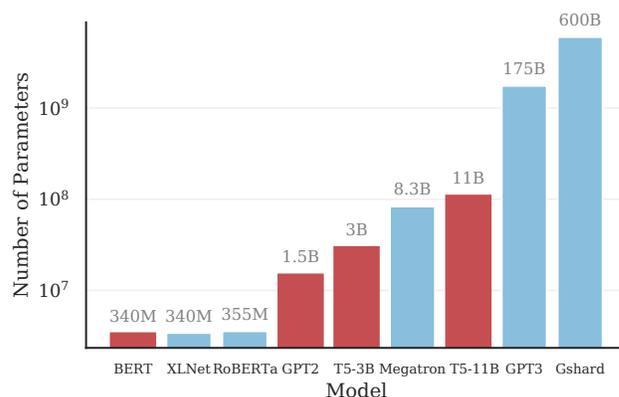


Figure 1: Model size trends. None of these models can be trained on a single NVIDIA RTX-2080-Ti commodity GPU due to memory requirements. Bars in red represent the models evaluated in this paper. FTPipe automatically transforms their sequential implementation into a multi-GPU one.

1 Introduction

Fine-tuning deep neural network (DNN) models is a technique commonly used to achieve state-of-the-art model quality on a wide range of tasks, such as question answering, text generation, translation, and more [46, 55, 56]. In fine-tuning, the model is not trained from scratch. Instead, a short training phase is performed on an already existing model, which has been pre-trained on large application-related datasets to obtain general domain knowledge [16]. By training on user-specific datasets, fine-tuning allows accommodating new inputs [11, 44, 60].

Fine-tuning is becoming increasingly important as models grow to billions of parameters. Since training such *giant* models from scratch is practical only on super-computer-scale

systems [48, 50, 60], a provider-scale pre-training, followed by short fine-tuning on user data, is the key to making the power of giant models broadly accessible [57].

Unfortunately, fine-tuning giant models on *low-cost commodity hardware* is still a significant challenge. Small memory sizes and slow inter-GPU communications are the main characteristics that set apart commodity GPUs from high-end devices, and they dramatically affect the training of giant models.

Small GPU memory is the key problem. For example, about 40 GBs are needed to train T5 model with 3B parameters for Recognizing Textual Entailment (RTE) [56] even with the smallest batch size of 1. This is larger than the memory size of any commodity GPU.

Existing approaches to cope with GPU memory constraints, such as sharding/swapping [45, 54], are ineffective due to slow PCIe communications. This problem is further exacerbated in fine-tuning tasks because the typically-small batch sizes do not allow for overhead mitigation using communication and computation overlapping. Therefore, using a single GPU, or a data-parallel multi-GPU approach for fine-tuning giant models are not viable options.

Model-parallel techniques allow fitting the model into memories of multiple GPUs. However, for commodity GPUs, the poor communication performance rules out the most commonly used Intra-Layer Model Parallelism approach, where each DNN layer is split among multiple GPUs. High volumes of all-to-all communications (AllReduce) with many synchronization points [48, 50] perform poorly over PCIe.

Pipeline-parallel execution exploits the depth dimension of DNN models [17, 37], distributing them across GPUs at a coarse granularity, i.e., DNN layers. A model is partitioned into multiple consecutive stages according to its topology. Each stage runs on a different GPU. During training, the input samples are streamed through the pipeline. This approach allows training giant models on multiple GPUs without the high communication costs of intra-layer parallelism, making it ideally suitable for commodity hardware.

However, existing methods have some drawbacks when applied to fine-tuning giant models.

GPipe [17] demonstrated the benefits of pipeline parallelization, but has pipeline bubbles, leading to low hardware utilization for small mini-batches typical for fine-tuning tasks [55]. The bubbles are caused by GPipe's *synchronous* approach whereby the next mini-batch of samples does not start before the previous one traverses the whole pipeline.

PipeDream [37] improved GPU utilization by introducing *asynchronous* training, where the mini-batches no longer wait for each other to complete. However, PipeDream stores multiple versions of the weights in GPU memory (*weight sharding*) to mitigate the effects of *weight staleness* [61] inherent to

asynchronous training, dramatically reducing the size of the models that can be trained.¹

Last, both GPipe and PipeDream may suffer from *poor load-balance* across GPUs in DNNs with a complex topology, such as giant encoder-decoder DNNs in Natural Language Processing (NLP). We claim that the load-balancing problem stems from the *coarse-grain scheduling decisions* dictated by the model topology constraints, i.e., the structure of the layers and inter-layer connectivity. Our key insight is that *relaxing these constraints when running fine-tuning tasks can mitigate the load imbalance of prior approaches and bring significant performance improvements, without model accuracy tax.*

We introduce **FTPipe**, a novel memory-efficient execution framework for pipelined fine-tuning of giant DNNs on commodity GPUs. Using FTPipe is easy: it *automatically* generates a functional performance-optimized pipeline-parallel multi-GPU version of a given sequential model implemented in PyTorch [39]. It enables the execution of models whose description does not adhere to the conventional sequences of coarse-grain layers.

FTPipe builds on two key observations:

Partitioning without DNN topology constraints. FTPipe optimizes the load balance by relaxing the model topology constraints when distributing it across the GPUs. Specifically, it introduces a novel *mixed-pipe* partitioning scheme which permits assigning any fine-grained combination of model operations to run on GPUs, *even when the assignment does not follow the original sequence of network-architecture layers*. This is in contrast to all prior approaches [17, 37] that use coarse-grain partitioning, with only adjacent layers scheduled to run together on the same GPU. The mixed-pipe scheme exploits the advantages of a new, previously unexplored, point in the trade-off between inter-GPU communications and load-balancing, prioritizing the latter. It enables efficient pipeline-parallel training of neural networks common in NLP, including language models with shared embedding ("tied weights") [20, 42] and giant Transformer-based encoder-decoder networks [53].

Fine-tuning large models is less sensitive to staleness. FTPipe fastest version employs an asynchronous training scheme which is prone to staleness [4, 5]. However, large models with pre-trained weights have properties that make training less sensitive to staleness. These include smooth and slowly-varying optimization trajectories [29], or small (and diminishing) learning rates. Thus, we achieve state-of-the-art results while avoiding costly staleness mitigation techniques.

We evaluate FTPipe on challenging fine-tuning tasks with three modern giant NLP models: T5-3B, T5-11B, GPT2 (1.5B), and BERT (340M). These models cannot be trained on a GPU with 11GB memory. To the best of our knowledge, FTPipe shows the first execution of T5-3B/T5-11B models on an asynchronous pipeline-parallel system, which have been

¹PipeDream still has staleness, with parameter consistency similar to ASGD [10], see Table 1.

particularly challenging due to the diverse computational requirements of different layers.

Our experiments on 8 NVIDIA RTX2080 Ti GPUs demonstrate significant performance benefits of FTPipe over GPipe, which is the only state-of-the-art pipeline execution framework that was able to train all the evaluated models on our hardware. For example, FTPipe is from 8% to $3\times$ faster when fine-tuning T5-3B, with the same accuracy. Mixed-pipe gains $1.08\times$ - $2.47\times$, and asynchronous training an additional $1.19\times$ - $2.98\times$ (results vary between datasets).

Compared to PipeDream, FTPipe trains significantly larger models and avoids partitioning solutions that do not fit in GPU memory. Further, FTPipe outperformed PipeDream by 13% when training BERT, after augmenting PipeDream implementation with checkpointing [7, 12].

Essentially, FTPipe generalizes the pipeline execution of former approaches to *dataflow-aware execution*, avoiding unnecessary communications and allowing more parallelism. Specifically, FTPipe stages can communicate with non-neighbors directly, without passing via adjacent stages. Thus, data is automatically loaded exactly where needed, and the stages can be concurrent in case the network graph allows it. As a result, model parameters and their gradients can be communicated thus making the execution of models with shared weights possible in pipelines.

In summary, our main contributions are as follows:

FTPipe. We present FTPipe, an automatic framework that transforms a sequential DNN implementation into a multi-GPU pipeline-parallel one, enabling fine-tuning of giant state-of-the-art neural networks on commodity GPUs,

Mixed-pipe. We present the fine-grain partitioning scheme which relaxes the model topology constraints when scheduling DNN computations on GPUs, thereby vastly improving load balance across the workers without additional communication overheads,

Evaluation. We evaluate FTPipe on challenging fine-tuning benchmarks, and giant transformers: GPT2 [43], BERT [11], and T5-3B/11B [44], consistently outperforming GPipe and PipeDream while attaining state-of-the-art accuracy.

2 Background

Giant models. We call *giant* those models which do not fit into the memory of a single accelerator during training.

Pipelined training. A pipeline is trained by computing gradients for each *mini-batch*, i.e., a subset of examples assigned together for gradient computations. A mini-batch can be further sliced into smaller *micro-batches* to fit into workers' memory. Micro-batches also help parallelize the mini-batch computation. For example, in GPipe, micro-batches are streamed through the pipeline one-by-one (Figure 2). GPipe attempts to parallelize execution by having a micro-batch proceed to the next pipeline stage while the next micro-batch enters the

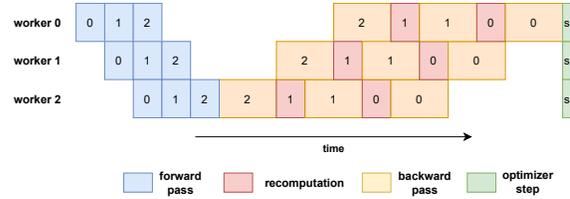


Figure 2: GPipe [17] synchronous training. Each rectangle (number is the micro-batch being processed) represents a pipeline stage running a certain task denoted by its color, except for the last optimizer step.

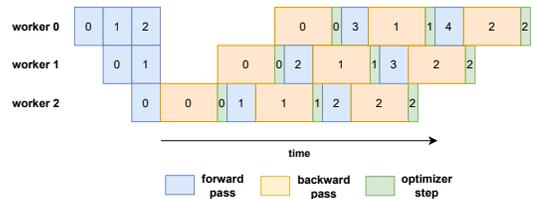


Figure 3: PipeDream [37] asynchronous training.

previous stage (aka intra-batch parallelism). However, GPipe *synchronously* updates the model parameters at the end of the mini-batch execution using the gradients accumulated from all micro-batches, a technique called *gradient accumulation*.

Staleness. An *asynchronous pipeline* as in PipeDream [37] (Figure 3) begins the forward pass of the next mini-batch without waiting for the previous one to finish (aka inter-batch parallelism). In an asynchronous pipeline, the next mini-batch might use old parameters in its forward pass and newly updated parameters on its backward pass, or old parameters in both passes. Pipeline with more stages may have more mini-batches executing concurrently, therefore a higher difference between old and new parameters. This problem, known as *weight staleness* (or simply staleness), was shown to introduce significant disturbances to the training process, deteriorating final model accuracy and, in extreme cases, even preventing convergence altogether [9, 62]. Several methods were proposed to mitigate staleness [4, 61].

3 Motivation

3.1 Importance of fine-tuning giant models

DNN models are constantly growing to achieve higher quality. This trend is particularly pronounced in Neural Language Processing models (Figure 1), where larger models achieve significantly better results [6, 17, 27]. However, such giant models are increasingly hard to train without access to supercomputer-scale resources. For example, training XLNet [60] of 340M parameters using a data set of 158GB required 5.5 days on 512 Google TPUs [23]. Therefore, many pre-trained models have been made publicly available to allow their use by those

who do not have access to such computing capabilities (e.g., NVIDIA Model catalog [1]).

From user point of view it is often desirable to further improve the pre-trained model and tailor it to a user-specific data set. The process, called *fine-tuning*, was shown to be effective across a wide range of tasks [16]. Fine-tuning operates on relatively small data sets and requires much less computing power to complete, compared to training a model from scratch.

Unfortunately, fine-tuning of giant models still require the whole model to be resident in GPU memory, which is a challenge for commodity GPUs. Attempts to fine-tune models by updating a subset of parameters [3, 15, 41] are not generally applicable and their results are usually worse than fine-tuning the entire model [44].

In summary, *fine-tuning of giant models poses a challenge that impedes their broader adoption.*

3.2 I/O benefits of pipeline model parallelism

Pipeline execution has the potential to fully overlap communication and computation in each pipeline stage, thus making the communication overhead negligible even on commodity hardware. There are two primary reasons:

Low communication volume. First, in pipeline-parallel training, as in any model-parallel training, GPUs communicate intermediate activations and activation-gradients, which, according to our experiments, are orders of magnitude smaller than parameter-gradients communicated in data-parallelism, and smaller than the number of state shards between workers. For example, a T5-3B model partitioned into 8 stages communicates a total of 456.4MB for each forward and backward pass across the whole pipeline (micro-batch size of 4, full precision). In comparison, in a data-parallel approach, each update requires collecting and aggregating 12GB of gradients per worker.

Overlapping computation and communication . Pipelines can overlap the communication with the next forward and/or backward passes, and sometimes also with the parameter update operation. This means that stages with large enough computation-to-communication ratio (for GPipe: ≥ 0.5) can completely overlap communication and computation. This is indeed the case for all giant models we consider in this work. For example, the aforementioned T5-3B achieves a per-worker average computation-to-communication ratio of 0.96 and 0.98 for the forward and backward passes respectively, even when communicated over PCIe.

3.3 Challenging load balance

GPipe and PipeDream obey model topology constraints when distributing the model across GPUs. Specifically, only adjacent layers are scheduled to run together on the same GPU.

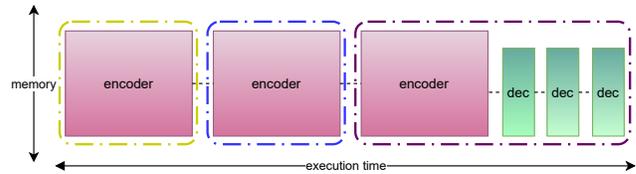


Figure 4: Partitioning unbalanced Encoder-Decoder models. Seq-pipes can assign only adjacent layers per GPU, thus are unable to partition the model for 3 GPUs in a balanced way (dashed lines). In contrast, Mixed-pipe can map one encoder and one decoder per GPU by assigning non-sequential layers to run together on the same GPU.

Such an approach, which we call sequential pipeline, or *seq-pipe*, fails to balance the DNN partitions across workers for some models.

One specific example of practical importance is the encoder-decoder architecture, widely used in NLP for sequence-to-sequence tasks [8, 28, 44, 53]. Consider the T5 Text-To-Text Transformer model, which is currently the state-of-the-art for many NLP tasks. When trained for question answering, the question is fed into the encoder, and the answer is fed into the decoder. In many cases, the answers are much shorter than the questions (e.g., yes/no questions). Thus, such training inputs cause a compute imbalance between encoders and decoders, since the computations in attention layers scale quadratically with the input sequence length. Furthermore, the number of parameters in both layer types is equally large, so grouping many “lightweight“ layers is not possible because it would exceed GPU memory. Thus, partitioning such models into a pipeline under strict model topology constraints (only adjacent layers allocated to the same GPU) is prone to severe compute and memory imbalance as illustrated in Figure 4.

PipeDream combines both pipeline- and data-parallelism, potentially solving the load imbalance by running compute-heavy stages on multiple GPUs using data parallel techniques. However, the amount of additional GPUs required may be impractical: consider the case of unbalanced encoder-decoders as in Figure 4, with N decoders, N encoders, and a computation-load ratio of k between encoder and decoder. A memory limit which allows placing no more than M encoders or decoders in the same device will imply that $\#GPUs = \frac{N}{M} \cdot K$ additional data parallel GPUs are required. In T5-3B for example, $M \leq 12$, $N = 24$, and K varies according to input sequence lengths of encoder and decoder (we measured $K = 5.5$ for two of our datasets where sequences were 512 and 4 respectively).

In FTPipe, we overcome the limitations of existing pipeline partitioning approaches and thus enable fast execution with state-of-the-art accuracy for challenging fine-tuning tasks on commodity GPUs.

4 FTPipe

Overview. FTPipe is a system for training giant models with limited resources. It automatically transforms models which do not fit into the memory of a single accelerator into a multi-GPU pipelined training construction, and runs them on our data-flow execution runtime. A computational graph of the DNN model is decomposed into topologically sorted sub-graphs, grouped into pipeline stages according to their graph distance from the output (called *depth*). The number of stages is determined according to the GPU assignment as discussed in detail below. Computations are then performed according to a work scheduler.

We now describe individual steps.

Tracing execution. Given a model and inputs, FTPipe first identifies the directed acyclic computational data-flow graph of the network. A *computational graph representation* of a neural network is a directed graph $G(V, E)$ where each intermediate computational operation is a node $v \in V$ and each edge $(u, v) = e \in E$ represents a connection according to computation order which is determined by the forward pass (with an indication of whether its also being used for the backward pass). We refer to indivisible blocks of execution as *basic blocks*. There are two granularity choices of basic blocks: full layers (In PyTorch, a layer is a software abstraction defined by ‘torch.nn.Module’ class), or individual operations (default setting). In the latter case, all operations visible in Python are traced, without splitting compiled C++ code or CUDA kernels.

Profiling. Each basic block is profiled to determine its memory consumption and execution time for each of its computational tasks, i.e., forward pass, recomputation (see Section 4.3) and backward pass. Aggregation of these values determines the block’s memory and computing requirements used in the block-to-GPU assignment step. Tensor sizes for activations and gradients are also recorded, and used to compute communication times, given a bandwidth parameter.

We note that tracing and profiling pose an engineering challenge on its own, since the models we discuss cannot run on a single device, hence, for example, cannot be profiled nor executed with dummy inputs to trace their execution graph. We solve this problem by decoupling execution and swapping layers, parameters, gradients, and activations in and out of host memory when needed.

Model partitioning. The objective of pipeline partitioning is to maximize throughput under memory and resource constraints. Maximizing throughput can also be described as minimizing the maximal *stage period* T_{max} among all the pipeline stages. We define T_i for stage i as

$$T_i = C + \max(0, T_{comm_{fwd}} - C) + \max(0, T_{comm_{bwd}} - C) \quad (1)$$

where $C = T_{compute_{fwd}} + T_{compute_{bwd}}$ and $T_{comm_{fwd}}, T_{comm_{bwd}}, T_{compute_{fwd}}, T_{compute_{bwd}}$ are the times of communication and computations for forward and backward

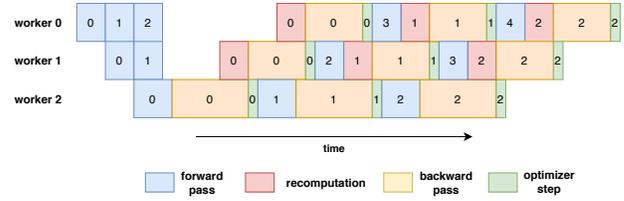


Figure 5: FTPipe asynchronous training. FTPipe uses checkpointing and recomputations to avoid storing multiple activations in memory. The last worker in the pipeline does not recompute, yet has the same period, load-balanced through profiling §4.3.

passes respectively (recomputation time is included in $T_{compute_{bwd}}$).

GPU assignment. The Mixed-pipe approach significantly inflates the search space of potential GPU allocations compared to Seq-pipe, to the point that - for the giant networks considered - an optimal, exhaustive assignment search (e.g., PipeDream’s) is infeasible. Thus, FTPipe’s GPU assignment takes a practical approach, combining efficient general graph partitioning and domain knowledge.

FTPipe employs several partitioning methods and let them compete, eventually selecting the best result according to its final throughput analysis via simulation. Among these schemes are (a) Mixed-pipe partitioning which searches mixed-pipe solutions as described in detail below, (b) PipeDream [37], which performs exhaustive search on the space of all Seq-pipes. (c) Acyclic [35, 36] which performs a greedy search for Seq-pipes, for which we changed the objective to optimize pipeline throughput, and combined with memory constraints. It is useful when an exhaustive search is too long to execute (for a 2000 nodes graph exhaustive search on Seq-pipes takes around 20 minutes). (d) Metis [24], a general graph partitioning scheme that optimizes communications under load balancing constraints. The output can be either a Seq-pipe or a Mixed-pipe. It does not optimize pipeline throughput directly.

Work scheduling. While FTPipe supports both synchronous and asynchronous pipeline work schedules, full fine-tuning acceleration benefits are obtained using an asynchronous work scheduler illustrated at Figure 5. FTPipe is a general pipelined data-flow execution. It supports concurrent stages, shared weights, and data staging. FTPipe’s work scheduling uses checkpointing with careful stage profiles (elaborated on in §4.3). In an asynchronous mode, for a stage of depth d , FTPipe first computes $d + 1$ forward passes for different micro-batches, filling the pipeline, then moves into alternating between backward and forward passes, modifying the model parameters after each mini-batch.

We next discuss Mixed-pipe partitioning and GPU assignment in detail.

4.1 Mixed-pipe model partitioning

Seq-pipe partitioning proposed in prior works assigns only adjacent DNN layers to every GPU in the pipeline. As a result, such partitioning *implicitly* optimizes for reduced communications across GPUs, at the expense of fewer opportunities to balance the load among them. Our intuition for Mixed-pipe is that for giant models and commodity hardware, keeping the computations balanced is sometimes more important than reducing inter-GPU communications. Indeed, for giant models, the overhead of pipeline data transfers is relatively small compared to the computation load per accelerator (See §3). We therefore posit that relaxing the DNN topology constraints and allowing smaller, multiple partitions per GPU may improve load balance, while still affording communication-computation overlap.

However, there are three main challenges in doing so: (a) it is possible that small partitions might increase both communications and GPU invocation overheads; (b) without topology constraints, an exhaustive search for optimal partitioning [37] is not feasible; (c) for asynchronous pipelines, higher-depth partitions may imply additional staleness, which may harm model quality. The Mixed-pipe partitioning scheme addresses these issues, and our evaluation empirically shows that the final model quality is maintained.

For efficient partitioning that mixes blocks of operations among the GPUs, it is imperative that changing the assignment of a block from one GPU to another does not increase communication penalty. To achieve this, notice that when the ratio between computation and communication is sufficiently high (see Equation 1), communication overhead is completely mitigated in FTPIPE by overlapping communication and computation. Thus, to enable dynamic placement of blocks in arbitrary GPUs, it is sufficient to ensure that the blocks have this property, called the *Communication-Computation Overlap* (CCO). The CCO property of a block of operations is easy to verify through profiling, taking into account both the forward edges (activations) and backward edges (gradients). We call a block with the CCO property a CCO block.

The Mixed-pipe partitioning algorithm receives a computational graph of a neural network, which specifies the (traced) basic computational blocks annotated with their memory, communication, and computing requirements. Then, partitioning proceeds in three steps as follows:

4.1.1 Step 1: Coarsening: create L pipeline stages

The graph is coarsened by contracting edges and merging their nodes to reduce its size (typically thousands of nodes) to L non-input stages where each stage fits in GPU memory. Throughout the process, a dynamic topological sort [40] is maintained to ensure that edge contractions do not create cycles.

Coarsening begins by eliminating nodes with either low computational load or high communication load, since both

are bad candidates to become CCO blocks. First, all constants and inputs are removed, later adding and duplicating them if needed to spare redundant communications (typically this process reduces the graph size by half). Second, nodes with a weight of 0 (typically CPU operations) are eliminated by merging them with their smallest neighbor (computation load-wise). Third, nodes for which the communication volume is above the 99th percentile are merged with their neighbors.

We proceed with coarsening with the goal of creating L CCO blocks, to become L pipeline stages. The algorithms involved are very fast, hence we try several coarsening heuristics, of which we describe two that gave the best results for giant networks we evaluated.

Coarsening by type. During tracing, the type and scope of each node are recorded according to its definition in the DNN code. By doing so, large groups of similar nodes can be naturally merged, following user high-level abstractions (e.g. merging all nodes inside ‘SelfAttention’ block of a Transformer [53]). This usually creates many equal-sized blocks of similar nodes, since giant networks typically contain repeated modules. Our experience shows that this process outputs the best partitioning of CCO blocks (and, consequently, pipeline stages) for giant networks.

Coarsening around centers. Adjacent CCO blocks are further joined together (notice that this maintains their CCO property) while obeying the memory constraint, until the block is sufficiently large to become a pipeline stage. One way to do this is to assign L “center blocks” and repeatedly merge their neighbors with them (obeying the memory constraint) in round-robin until no non-center block is left. This procedure may repeat several times with different choices of centers until a good packing of stages is obtained.

Choosing L . L is chosen at a sweet spot of the tradeoff between large L , which enables better load balancing in Step 2, and small L , which is better staleness-wise. To this end, we try several values of L (e.g. $L = 2P, 3P$) where P is the number of GPUs, and take the option yielding the best pipeline throughput (measured at step 3 below), where L does not exceed a pre-defined upperbound which can be found through experimentation.

4.1.2 Step 2: Load balancing

L stages are assigned to P GPUs in a way that optimizes load-balancing while ensuring that tasks allocated to the same GPU fit in its memory.

The assignment to GPUs can be seen as a classical multiprocessor scheduling problem [26]. While the original problem does not target the pipeline execution setup, in practice, and when communications do not add overhead (the CCO property), the optimal multiprocessor schedule is the one with a balanced distribution of tasks among the processors, meeting the goal of the pipeline assignment. Therefore, we apply the broadly used Longest-Processing Time-First heuristics: the L

CCO stages are sorted in a descending order of their computation load, and are assigned to the next least busy GPU with enough memory to run them.

Pipeline stages inside each GPU are created according to the topological order of the stages in the computational graph. A simple algorithm ensures that the depth of the stages is minimized: merge connected components inside each GPU as long as this does not create a cycle with other components. During execution, a stage is invoked by the GPU when its inputs are made available by completion of execution of all previous stages.

4.1.3 Step 3: Refinement

We perform fine-grain tuning of the load balance achieved in Step 2. First, the L stages are un-coarsened into their basic blocks. Then, within each pipeline stage, we greedily find blocks which, if moved to an adjacent stage in the pipeline, can improve the throughput, or lower communication or improve load balance (given also the memory constraints).

The complexity of Mixed-pipe partitioning is dominated by the complexity of coarsening which is $O(N^2d)$, where d is the average node degree. Assuming we try c different coarsening strategies (e.g., initialization of centers, different choices of L , etc.), the overall complexity is $O(cN^2d)$.

4.2 Fine-tuning with staleness

Asynchronous pipelines allow faster execution. However, asynchrony introduces staleness. The key problem with staleness is that it can potentially harm final model quality. Note, however, that this is analogous to large batch training [47] which proved to be a popular and useful technique when applicable. In such cases, empirical evidence of acceleration while maintaining model quality is of practical importance even when shown only for some specific (yet important) cases.

Some staleness mitigation techniques introduce high overheads. For example, weight stashing, applied in PipeDream, causes multiple versions of model weights to be stored for the entire course of the pipeline round-trip. This implies that in a pipeline with K workers, the first worker stores up to K versions of weights, which might effectively nullify the memory benefits of model partitioning across the GPUs.

In the case of pipeline execution of fine-tuning tasks, however, we observe that staleness *does not* have a major deteriorating effect on the training quality. There could be several reasons for this phenomenon: (a) Staleness is higher in the initial phase of the training [18], while in fine-tuning we start with pre-trained weights; (b) Fine-tuning usually uses lower learning rates, hence smaller staleness gaps [4]. (c) Momentum [51] exacerbates staleness, but many fine-tuning tasks do not use momentum. Furthermore, the part of staleness caused by momentum can be mitigated using gradient accumulation and momentum weight prediction [13], both can be employed

Setting	Loss	Forward	Backward
Synchronous	θ_t	θ_t	θ_t
Asynchronous	θ_{t-s}	θ_{t-s}	θ_t
PipeDream	θ_{t-s}	θ_{t-s}	θ_{t-s}
FTPipe	θ_{t-s}	θ_t	θ_t

Table 1: Parameter versions during different phases of back-propagation, under different pipeline execution scenarios. θ_j denotes parameters after j optimization steps. A stale parameter with delay of s is denoted θ_{t-s} . Notice that the pass used for loss calculation ('loss') and the pass used to compute activations for backpropagation ('forward pass') induce two separate calculations.

by FTPipe with no memory overhead and a small computational overhead ($< 5\%$, measured on Bert and GPT2). (d) Large models appear to have smooth and slowly varying loss functions [29], meaning that for large models, the stale loss can be a close-enough approximation to the actual loss. Indeed, as can be observed in the experiments Section 5, FTPipe achieves state-of-the-art results with an asynchronous pipeline of up to 16 stages, while avoiding the memory overheads of staleness mitigation implemented in prior works [37].

4.2.1 Checkpointing and recomputation

Checkpointing [7, 12] is a method to reduce memory by keeping only a small state (essentially, partition-border activations and the seed of the random number generator) rather than the whole computation graph. Checkpointing takes place during the forward pass while computing the loss. In this work, we experimented with checkpointing at pipeline stage borders similar to GPipe (In principle, checkpointing can be used at higher granularity). During the backward pass, recomputation reconstructs the parts of the computation graph required for backpropagation.

Checkpointing is used by FTPipe for saving memory during training, but it also mitigates staleness in two ways: First, during backpropagation, recomputation implies that the gradients are computed on up-to-date parameters (but using a stale loss that was computed in a forward pass on an older set of parameters, Table 1). Second, more computations are shifted toward the end of the pipeline where staleness is lower as a result of the last pipeline stage not recomputing, which makes it possible to increase its computational load by around 33%.

4.3 Profiling

In FTPipe's asynchronous work scheduler (Figure 5) there are two types of stages in the pipeline: (A) the last stage, which does not recompute (B) other stages, which do recompute.

For this reason, the execution of the last stage is faster by approximately 33% [17]. Hence, for precisely estimating the load, FTPipe profiles blocks both with and without recomputation. When a block is assigned to the last stage, it uses its profile without recomputations, and vice versa. However, the partitioning itself needs to know which profile to use when the block goes through the coarsening process, and this profile may eventually end up being the wrong one. Thus, several iterations of the partitioning algorithm may be needed to ensure that the correct profile is selected.

5 Evaluation

In this section, we evaluate the performance of FTPipe.

Summary. First, we show that FTPipe significantly improves the existing methods for fine-tuning of giant models.

Second, we show the benefit of Mixed-pipe for both synchronous and asynchronous pipelines. In both cases, despite communicating more and assigning non-consecutive layers of the trained model to workers, we show that Mixed-pipe accelerates execution compared to a Seq-pipe baseline.

Third, we show the benefits of fine-tuning giant models asynchronously, despite the fact that asynchrony introduces staleness. We do so by comparing FTPipe asynchronous pipeline to GPipe synchronous pipeline when both are employed to train the same model using the same partitioning. Our results show that FTPipe asynchronous pipeline is faster to achieve the same top-accuracy of GPipe, across different architectures, datasets, optimizers, and model sizes. Furthermore, our results empirically prove that the additional staleness generated by the increased number of pipeline stages in Mixed-pipe does not degrade its final accuracy.

Implementation. FTPipe has two main components: (a) an automatic neural network partitioning and assignment which builds a data-flow, and (b) a pipelined data-flow execution runtime that executes the partitioned model on GPUs and automatically handles work scheduling and inter-GPU communications. We use CUDA-Aware OpenMPI for inter-GPU communications. Our implementation accepts as the input a neural network implemented in PyTorch [39] (Python API) and the representative training inputs for this network. A partitioned model is automatically generated with everything necessary for our runtime to run it.²

5.1 Experimental setup

Hardware. We use a server with 8 RTX2080-Ti GPUs each with 11GB memory, connected via PCIe-III, 64-bit Ubuntu 18.04 with CUDA toolkit 10.2 and cuDNN v7.6.5. We note that RTX2080 GPUs are considered 7x more cost-effective than V100 [14].

²The source code of FTPipe is available at: <https://github.com/saareliad/FTPipe>

Models and training methodology. We used three different model architectures: Bert [11], GPT2 [43] and T5 [44]. These models represent the typical kinds of NLP architectures used for fine-tuning: Bert is encoder-only, GPT2 is decoder-only, and T5 is encoder-decoder. We took the PyTorch implementations of the models and pretrained weights from HuggingFace [57].

For each combination of (dataset, model, pipeline, partitioning), we chose the number of micro-batches that achieved the best throughput while keeping the mini-batch size constant. For example, GPipe prefers a higher number of micro-batches for a given mini-batch size to increase its pipeline parallelization level. In contrast, FTPipe prefers a lower number of micro-batches as it uses inter-mini-batch parallelism.

Exact hyper parameters are reported in [Appendix A](#).

Tasks and datasets. We use five different learning tasks and six datasets: Natural Language Inference (NLI) using RTE, Word Sense Disambiguation using WiC, Question answering using SQuAD and MultiRC, Boolean Question Answering using BoolQ and Language Modeling using WikiText2 [34, 46, 55, 56].

Our choices are dictated by the following criteria. We choose the task for a given model if that model is known to achieve state-of-the-art results, implying that smaller models are inferior. For example, RTE, WiC, BoolQ, MultiRC were chosen since T5 improved their accuracy considerably. GPT2 for WikiText2 achieves state-of-the-art results. Our fine-tuning improved the advertised results by 6.32 perplexity points on the test set.

We fine-tuned all models and datasets, achieving accuracy comparable to the top published results.

Baselines. Our choice of the baseline is restricted because of the memory requirements of giant models. In particular, neither Single-GPU, data-parallelism [30] nor PipeDream [37] without checkpointing meet these requirements. Hence, we apply checkpointing to PipeDream. Mesh-TensorFlow model-parallel framework [48] failed to run T5-3B, running out of memory, even with checkpointing, FP16, and micro-batch size of 1. This framework was originally used to run T5 on a TPU cluster, but the commodity hardware restrictions impede its use. In general, however, PipeDream already demonstrated the throughput benefits of pipelined execution over the model-parallel one, which applies to our work too.

GPipe is the only framework that successfully runs all the evaluated models. We used synchronous execution with GPipe to set the target accuracy in our experiments.

5.2 End-to-end evaluation

Table 2 shows that *FTPipe outperforms GPipe for all the cases in terms of time-to-top-accuracy*. As expected, the main

Table 2: Summary of the comparison of FTPIPE with original GPipe Seq-pipe (synchronous pipeline, one stage per GPU) over 8 GPUs. TTA is Time to Top Accuracy. The top accuracy is according to GPipe’s best results. Speedup is a geomean of four runs with random seeds. Mixed-pipe and Seq-pipe are denoted as “mixed” and “sequential” partitioning respectively. WSD is Word Sense Disambiguation, NLI is Natural Language Inference, QA is Question Answering, and LM is Language Modeling.

Model	Size	Task	Dataset	Accuracy	Partitioning	Pipeline	Speedup over GPipe Epoch time	Speedup over GPipe TTA
T5	3B	WSD	WiC	74.92%	mixed	Async	2.48×	³ 11.54×
						Sync	1.19×	1.19×
		NLI	RTE	90.97%	mixed	Async	2.71×	4.1×
						Sync	1.2×	1.2×
		QA	BoolQ	89.05%	mixed	Async	2.13×	2.88×
QA	MultiRC	85.6 F1, 59.3 EM	mixed	Sync	1.08×	1.08×		
GPT2	1.5B	LM	WikiText2	12.02 perplexity	sequential	Async	1.6×	1.6×
Bert	340M	QA	Squad	93.3 F1, 87.2 EM	sequential	Async	2.04×	2.04×

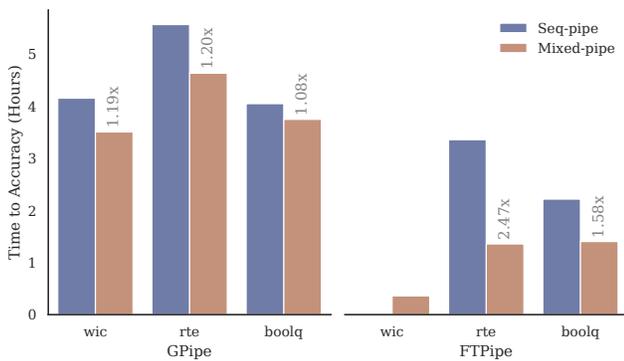


Figure 6: Mixed-pipe speedup over Seq-pipe for reaching top accuracy with T5-3B on three datasets and two pipelines: GPipe (synchronous) and FTPIPE (asynchronous).

speedup is due to faster epoch time, but it is not the only factor.

To explain this phenomenon, Figure 7 shows an example of FTPIPE training accuracy over time for T5-3B on RTE. We observe that both systems achieve 40% or higher accuracy at about the same time, despite the extra epochs needed for FTPIPE. Nevertheless, FTPIPE is much faster in the remaining top 60%. The reason is FTPIPE staleness, which impacts the training process in the beginning, yet diminishes and becomes negligible as the learning steps shorten [18].

In the following sections, we analyze the two factors contributing to acceleration, namely Mixed-pipe and asynchronous execution.

³ FTPIPE required fewer epochs to achieve the top accuracy of GPipe. The speedups vary from 3× to 40× over 4 runs with different seeds.

5.3 Effect of mixed-pipe partitioning

Figure 6 compares the contribution of our partitioning scheme by evaluating different schemes separately for synchronous and asynchronous pipelines for the T5 model on three different tasks and four datasets. Mixed-pipe runs 16 stages (two per GPU). Seq-pipe runs eight stages. We observe that Mixed-pipe is superior to Seq-pipe across all the experiments. We observed that Mixed-pipe balances GPU memory occupancy across GPUs for both synchronous and asynchronous pipelines, keeping more free memory available, thus allowing a bigger micro-batch size. Most importantly, Mixed-pipe improves both synchronous and asynchronous. We note that while the inter-GPU communication volume in Mixed-pipe is higher than in Seq-pipe, 2.6× for MultiRC and BoolQ, 2.5× for RTE and 3.31× for WiC, the benefits of improved load balance outweigh the associated overheads.

The benefits and trade-offs Mixed-pipe introduces differs for synchronous and asynchronous pipelines:

Synchronous pipeline. For the synchronous setting, using Mixed-pipe can only improve performance and does not affect accuracy. Figure 6 shows that Mixed-pipe improves the time-to-accuracy of GPipe up to 1.2×.

Asynchronous pipeline. For FTPIPE asynchronous pipeline, Mixed-pipe may harm accuracy since it adds more pipeline stages, which add staleness. However, in practice, accuracy is not affected. Figure 6 shows that Mixed-pipe both accelerates execution and achieves the same final accuracy result of Seq-pipe. Figure 7 provides the explanation for this phenomena: staleness affects mainly the beginning of the computation, up to the point of reaching 40% of the target accuracy. In this first part of the computation, staleness causes Mixed-pipe to execute more epochs than Seq-pipe, yet they both run at a similar wall-clock speed because Mixed-pipe is

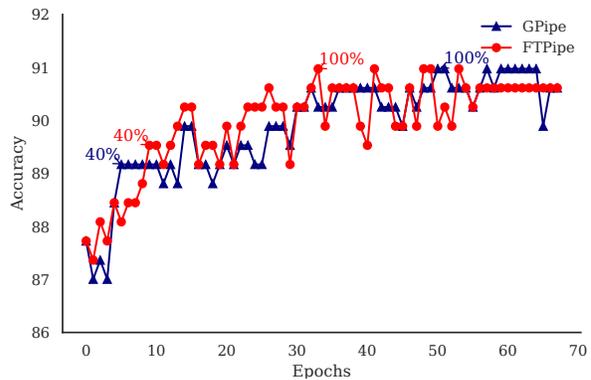
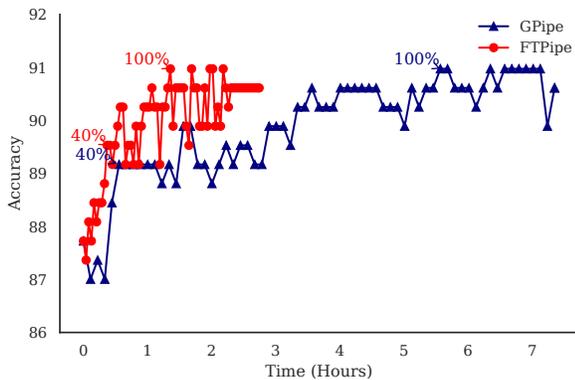


Figure 7: FTPipe acceleration over GPipe for fine-tuning T5-3B with Glue RTE dataset. FTPipe uses Mixed-pipe with 16 stages. GPipe is a synchronous Seq-pipe with 8 stages. We denote the earliest times when at least 40% and 100% of the top accuracy is achieved. Notice that FTPipe is much faster in achieving the last top 60%.

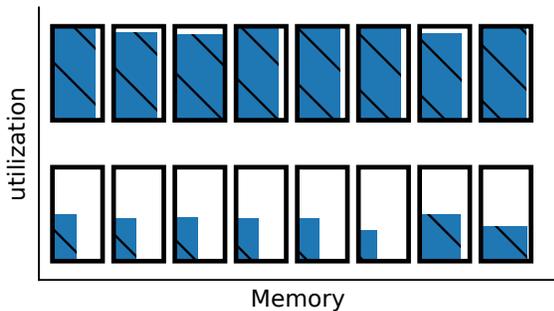


Figure 8: Mixed-pipe load balancing visualization for asynchronous pipelines with T5-3B model on the WiC dataset. Each box is a GPU whose width represents memory consumption and fill represents utilization. Top: Mixed-pipe. Bottom: Seq-pipe.

much faster per epoch. In contrast, during the remaining part of the computations from 40% to 100% of the target accuracy, Mixed-pipe enjoys significant speedup over Seq-pipe with negligible (if at all) effect of staleness as the learning steps gradually shorten [18].

5.3.1 Load balancing analysis

In Figure 8, we present the load balancing analysis of Mixed-pipe, using T5 as a representative example.

As illustrated in Figure 4, the T5 encoder-decoder architecture with unbalanced inputs sequences for encoder and decoder poses a problem for Seq-pipe. In our experiments, Seq-pipe with memory-unaware partitioning methods resulted in out of memory run-time errors, whereas Seq-pipe with memory-aware partitioning methods resulted in computa-

tional load imbalance. In contrast, Mixed-pipe achieves a much better balance both for memory occupancy and computational load.

In summary, *Mixed-pipe improves the performance over Seq-pipe due to two main factors: (1) better computational load balance and (2) better memory balance, thereby enabling larger micro-batch size.*

5.4 Effective fine-tuning with staleness

Table 3 summarises our results for three different model sizes and five different tasks. Each task used the same model partitioning to train GPipe (synchronous) and FTPipe (asynchronous). Note that these results differ from those in Table 2, because here we use the Mixed-pipe partitioning scheme for both FTPipe and GPipe.

In all experiments, FTPipe achieved the same or higher accuracy than GPipe. We note that MultiRC, Squad and WiC achieved slightly better results with FTPipe (85.99 F1 60.44 EM; 93.47 F1 87.38 EM; and 75.54 accuracy receptively).

5.5 FTPipe vs PipeDream

We implemented PipeDream partitioning based on their open-source GitHub code. To focus on conceptual evaluation, the partitioning algorithm is implemented on top of the technically superior tracing, profiling, compilation, and runtime system of FTPipe, which can handle giant models. Furthermore, it is necessary since only FTPipe supports sharing weights.

T5-3B: PipeDream partitioning of T5-3B for 8 GPUs required around 20 minutes and yielded an infeasible solution with hybrid data- and pipeline-parallel stages, some of which do not fit into GPU memory. In particular, lacking the way to specify memory constraints, PipeDream ended up with more than 1.48B parameters in the last stage (a single 32GB V100 GPU can handle only about 1.3B parameters [45]). It also produced

Table 3: FTPipe vs GPipe time-to-accuracy, using the *same* Mixed-pipe partitioning. Top accuracy is set by GPipe best results.

Dataset	Model	Parti- -tioning	Top Accuracy	Speedup	
				Epoch	TTA
wic	T5-3B	mixed	74.92%	2.09×	9.74×
rte	T5-3B	mixed	90.97%	2.25×	3.41×
boolq	T5-3B	mixed	89.05%	2.23×	2.67×
multirc	T5-3B	mixed	85.6 F1 59.3 EM	2.98×	2.98×
Wiki -Text2	GPT2- 1.5B	seq	12.02 perplexity	1.61×	1.61×
Squad	Bert- 340M	seq	93.3 F1 87.2 EM	2.04×	2.04×

an infeasible solution (OOM) when forced to create a simple pipeline (no data-parallel stages). Note that a pipeline with weight stashing and without gradient accumulation would not be able to train T5-3B on 11GB GPUs, even if more GPUs were used, as explained in §4.2.

GPT2-1.5B PipeDream failed to run GPT2-1.5B with 8 GPUs even with checkpointing.

BERT-large: When profiling with micro-batch of size 24, and sequence length 384 (the suggested hyperparameters for SQuAD [11]), PipeDream outputs a purely data-parallel solution (no pipeline) for 2, 4, and 8 GPUs. Such a solution does not fit RTX2080-Ti GPUs even with batch size 1. Only at micro-batch size 1 PipeDream succeeded in creating a working pipeline for 2 GPUs. For fine-tuning SQuAD with 2 GPUs, FTPipe was 12.7% faster than PipeDream in achieving the same accuracy over 5 seeds (std was 1%), attributed to the enhanced profiling of FTPipe.

For 4 GPUs, FTPipe was 4% faster than PipeDream pipeline⁴. For 8 GPUs, both PipeDream (pipeline) and FTPipe achieve comparable accuracy at comparable times, since the advantage of FTPipe profiling diminishes as the number of pipeline workers increases.

5.6 Layers-graph vs Operators-graph

We evaluate the contribution of fine-grain partitioning to the end-to-end performance of T5-3B using WiC dataset.

For Mixed-pipe partitioning the operators-graph results in epoch speedup of 2.2% for the asynchronous pipeline and 1.6% for the synchronous pipeline, compared to the cases of partitioning the layers-graph (standard deviation less than 0.01%). This implies that the primary source of speedup for

⁴We noticed that PipeDream models data-parallel communication as *completely* concurrent to computation, but in practice, 30% of the communications do not overlap [30]. Changing the modeling led PipeDream to output a pipeline solution.

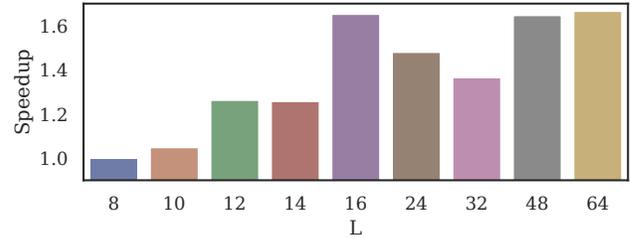


Figure 9: Mixed-pipe performance for different values of L .

this case comes from relaxing the neural network topology when mapping layers (or operators) to GPUs.

However, this is not always the case. We build a small version of T5-11B, with 1 encoder layers and 1 decoder layer (535M parameters), and compare the partitions for $P = 2$ GPUs, once at the granularity of whole layers, and once at the granularity of operators. The latter achieves 45% better min-max-stage-time objective in our simulation.

We also note that the layers-graph is not explicitly expressed in some cases, leaving the operator-granular partitioning as the only option. This is the case, for example, when importing a graph defined in ONNX [2] for further fine-tuning before deployment, or if a programmer does not use the layer software abstraction when defining the network (in PyTorch: ‘torch.nn.Module’).

5.7 Mixed-pipe sensitivity to L

We explore the sensitivity of the Mixed-pipe performance to the choice of L .

In Figure 9 we show the partitioning objective speedup for T5-3B (BoolQ) for $P = 8$ GPUs when using different L for partitioning, with FTPipe’s asynchronous scheduler (Figure 5). We observe that that the sweetspot is at $L = 16$.

5.8 Mixed-pipe vs Seq-pipe for larger models

In this experiment, we partitioned the T5-11B model (BoolQ dataset) to four 32GB V100 GPUs using both Mixed-pipe and Seq-pipe. Mixed-pipe partitioning allowed training the model using 7 workers. Unfortunately, Seq-pipe partitioning resulted in an out-of-memory error. This experiment highlights the importance of better memory balancing among the nodes to allow fine-tuning huge models.

6 Related Work

Parallel, memory efficient methods for training giant neural networks fall into three categories: Sharded Data Parallelism [45], Intra Layer Model Parallelism [48, 50], and synchronous pipeline parallelism with checkpointing [17]. The communication volumes of intra-layer model-parallelism and

shared data-parallelism cannot be overlapped entirely and are too high for commodity interconnects. GPipe [17], similar to sharded data-parallelism, can be made efficient and hide synchronization overheads when using large mini-batches with many micro-batches. However, this is not generally applicable to the batch size used with fine-tuning. Compressing the communication of data-parallelism [32] can significantly reduce its communication volume, making it more suitable for commodity interconnects. For giant networks, however, a combination with another memory reduction technique is required.

In another line of works [18, 19, 21, 59], the backward pass is decoupled and is performed in parallel, but the forward pass is sequential. In Ouroboros [59], shared embedding (aka Tied Weights [20, 42]) of small Transformers were manually placed on the same GPU. In FTPipe, shared weights can also be placed on different GPUs, and it is done with automatic partitioning and with pipelining where communication is overlapped.

PipeDream [37] solves pipeline imbalance problems by incorporating data-parallelism with seq-pipes, allowing a different number of data-parallel GPUs per pipeline stage. This solution requires more GPUs than Mixed-pipe to balance some architectures (see §3.3).

Our experiments with giant models show that the exhaustive search of PipeDream can take a long time for large computational graphs (where FTPipe search took few seconds) and is infeasible for Mixed-pipe. METIS [24] rapidly partitions large computational graphs by multilevel graph partitioning. However, it is focused on optimizing edge-cut under load balance constraints and not on the pipeline throughput. When applied to pipelines, it sometimes unnecessarily creates additional small stages, which Mixed-pipe bounds from above by L .

Concurrently and independently of our work, Tarnawski et al [52] considered searching for non-contiguous pipeline solutions with Integer Programming. They did not model communication and computation overlap and did not limit the number of stages L , thus also ignoring staleness. They did not find the optimal solution and stopped the search after 20 minutes for models smaller than those considered here.

Several asynchronous model parallelization methods have been recently proposed [38, 58]. These works use Seq-pipes and would suffer from imbalance problems solved by Mixed-pipe. FTPipe could be used in heterogeneous systems [38], but the large search space requires additional heuristics. GEMS [22] improve synchronous pipelines in case multiple replicas could fit in the accelerators. Li et al. [31] proposed token-level parallelism which applies to causal language models (only GPT2 in our experiments), and is complementary to FTPipe.

7 Conclusion

Fine-tuning has the potential to bring the power of huge neural networks to users who do not possess high performance compute resources. In this paper we survey the challenges introduced by this paradigm and take a big step towards solving them by enabling fine tuning on affordable commodity hardware. Our future work will show that FTPipe can scale beyond single-machine boundaries, can achieve higher efficiency by incorporating data-parallel components, and can bring value to non-NLP communities. We hope that the ideas underlying Mixed-pipe can apply beyond fine-tuning, to training in general.

8 Acknowledgements

We thank our shepherd Tim Harris for the insightful comments that helped us improve the paper. This work was supported in part by the Israeli Ministry of Science, Technology, and Space, the Israeli Science Foundation, and by The Hasso Plattner Institute at the Technion.

References

- [1] Nvidia ngc. <https://NGC.nvidia.com>.
- [2] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. <https://github.com/onnx/onnx>, 2019.
- [3] Ankur Bapna and Orhan Firat. Simple, scalable adaptation for neural machine translation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1538–1548, Hong Kong, China, November 2019. Association for Computational Linguistics.
- [4] Saar Barkai, Ido Hakimi, and Assaf Schuster. Gap-aware mitigation of gradient staleness. In *International Conference on Learning Representations*, 2020.
- [5] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.
- [6] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Nee-lakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.
- [7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

- [8] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.
- [9] Wei Dai, Yi Zhou, Nanqing Dong, Hao Zhang, and Eric Xing. Toward understanding the impact of staleness in distributed machine learning. In *International Conference on Learning Representations*, 2019.
- [10] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc’auelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.
- [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, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.
- [12] Andreas Griewank and Andrea Walther. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.
- [13] Ido Hakimi, Saar Barkai, Moshe Gabel, and Assaf Schuster. Taming momentum in a distributed asynchronous environment. *arXiv preprint arXiv:1907.11612*, 2019.
- [14] Dodi Heryadi and Scott Hampton. Characterizing performance improvement of gpus. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*, pages 1–5. 2019.
- [15] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. volume 97 of *Proceedings of Machine Learning Research*, pages 2790–2799, Long Beach, California, USA, 09–15 Jun 2019. PMLR.
- [16] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 328–339, Melbourne, Australia, July 2018. Association for Computational Linguistics.
- [17] 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. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019.
- [18] Zhouyuan Huo, Bin Gu, and Heng Huang. Training neural networks using features replay. In *Advances in Neural Information Processing Systems*, pages 6659–6668, 2018.
- [19] Zhouyuan Huo, Bin Gu, qian Yang, and Heng Huang. Decoupled parallel backpropagation with convergence guarantee. volume 80 of *Proceedings of Machine Learning Research*, pages 2098–2106, Stockholmssmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.
- [20] Hakan Inan, Khashayar Khosravi, and Richard Socher. Tying word vectors and word classifiers: A loss framework for language modeling. *arXiv preprint arXiv:1611.01462*, 2016.
- [21] Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, David Silver, and Koray Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1627–1635. JMLR. org, 2017.
- [22] Arpan Jain, Ammar Awan, Asmaa Aljuhani, Jahanzeb Hashmi, Quentin Anthony, Hari Subramoni, Dhambaleswar Panda, Raghu Machiraju, and Anil Parwani. Gems: Gpu-enabled memory-aware model-parallelism system for distributed dnn training. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 621–635. IEEE Computer Society, 2020.
- [23] 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 *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.
- [24] George Karypis and Vipin Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.
- [25] Diederik P Kingma and Lei Ba. J. ADAM: a method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

- [26] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.
- [27] 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.
- [28] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.
- [29] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. *Advances in neural information processing systems*, 31:6389–6399, 2018.
- [30] 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. *arXiv preprint arXiv:2006.15704*, 2020.
- [31] 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. *arXiv preprint arXiv:2102.07988*, 2021.
- [32] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018.
- [33] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.
- [34] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.
- [35] Orlando Moreira, Merten Popp, and Christian Schulz. Graph partitioning with acyclicity constraints. *arXiv preprint arXiv:1704.00705*, 2017.
- [36] Orlando Moreira, Merten Popp, and Christian Schulz. Evolutionary multi-level acyclic graph partitioning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 332–339, 2018.
- [37] 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.
- [38] 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. *arXiv preprint arXiv:2005.14038*, 2020.
- [39] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, 2019.
- [40] David J Pearce and Paul HJ Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *Journal of Experimental Algorithmics (JEA)*, 11:1–7, 2007.
- [41] Matthew E. Peters, Sebastian Ruder, and Noah A. Smith. To tune or not to tune? adapting pretrained representations to diverse tasks. In *Proceedings of the 4th Workshop on Representation Learning for NLP (RepLanLP-2019)*, pages 7–14, Florence, Italy, August 2019. Association for Computational Linguistics.
- [42] Ofir Press and Lior Wolf. Using the output embedding to improve language models. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 157–163, Valencia, Spain, April 2017. Association for Computational Linguistics.
- [43] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 2019.
- [44] 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. *Journal of Machine Learning Research*, 21(140):1–67, 2020.
- [45] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimization towards training a trillion parameter models. *arXiv preprint arXiv:1910.02054*, 2019.
- [46] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine

- comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, November 2016. Association for Computational Linguistics.
- [47] Christopher J Shallue, Jaehoon Lee, Joe Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. M. *arXiv preprint arXiv:1811.03600*, 2018.
- [48] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, pages 10414–10423, 2018.
- [49] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. volume 80 of *Proceedings of Machine Learning Research*, pages 4596–4604, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.
- [50] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- [51] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.
- [52] Jakub Tarnawski, Amar Phanishayee, Nikhil R. Devanur, Divya Mahajan, and Fanny Nina Paravecino. Efficient algorithms for device placement of dnn graph operators. In *Neural Information Processing Systems (NeurIPS 2020)*, December 2020.
- [53] 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.
- [54] Mohamed Wahib, Haoyu Zhang, Truong Thao Nguyen, Aleksandr Drozd, Jens Domke, Lingqi Zhang, Ryousei Takano, and Satoshi Matsuoka. Scaling distributed deep learning workloads beyond the memory capacity with karma. *arXiv preprint arXiv:2008.11421*, 2020.
- [55] Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems. In *Advances in Neural Information Processing Systems*, pages 3266–3280, 2019.
- [56] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, Brussels, Belgium, November 2018. Association for Computational Linguistics.
- [57] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, R’emi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface’s transformers: State-of-the-art natural language processing. *ArXiv*, abs/1910.03771, 2019.
- [58] An Xu, Zhouyuan Huo, and Heng Huang. On the acceleration of deep learning model parallelism with staleness. In *The IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.
- [59] Qian Yang, Zhouyuan Huo, Wenlin Wang, and Lawrence Carin. Ouroboros: On accelerating training of transformer-based language models. In *Advances in Neural Information Processing Systems*, pages 5520–5530, 2019.
- [60] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems*, pages 5753–5763, 2019.
- [61] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. Staleness-aware async-SGD for distributed deep learning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, 2016.
- [62] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhi-Ming Ma, and Tie-Yan Liu. Asynchronous stochastic gradient descent with delay compensation. In *International Conference on Machine Learning*, pages 4120–4129. PMLR, 2017.

Table 4: Hyper-parameters we used. Accum stands for gradient accumulation steps (micro batches). Batch is mini-batch size. Max steps are the max steps the model was trained for.

Dataset	Pipeline	Batch	Accum	Max steps
squad	ftpipe-seq	24	1	2 epochs
	gpipe-seq	24	8	2 epochs
wikitext2	ftpipe/gpipe-seq	8	8	1 epoch
rte	ftpipe-seq	40	10	4200
	ftpipe-mixed	40	5	4200
	gpipe-seq/mixed	40	10	4200
wic	ftpipe-seq	128	4	17000
	ftpipe-mixed	128	2	17000
	gpipe-seq/mixed	128	8	17000
boolq	ftpipe-seq	20	10	3200
	fpipe-mixed	20	5	3200
	gpipe-seq/mixed	20	10	3200
multirc	ftpipe-seq	8	4	17000
	fpipe-mixed	8	2	17000
	gpipe-seq/mixed	8	8	17000

Appendix A Hyper-parameters

Table 4 lists the experiments’ hyper-parameters. A checkpoint was saved after every epoch and for WiC every 100 steps. In the T5 and GPT2 experiments, we utilized gradient accumulation to achieve the desired batch size.

T5 experiments used Adafactor optimizer [49] with learning rate of 0.01 and a warm-up of approximately 6% of total fine-tuning steps.

GPT2 experiments used AdamW [33] optimizer with weight decay of 0.01 and a learning rate of 5e-5, decreasing linearly to zero. First and second moment coefficients are 0.9 and 0.999, respectively. We used a mini-batch size of 8 and max sequence length of 1024. We fine-tuned GPT2 for one epoch since further fine-tuning caused over-fitting. Due to large memory consumption caused by the large sequence-length (1024), we used a batch size of 1.

BERT experiments used Adam [25] optimizer with first and second moment coefficients of 0.9 and 0.999 respectively, a learning rate of 3e-5 decreasing linearly to zero. We trained for 2 epochs with mini-batch size of 24 and max sequence length of 384. We loaded weights pre-trained with whole-word-masking.

INFaaS: Automated *Model-less* Inference Serving

Francisco Romero*, Qian Li*, Neeraja J. Yadwadkar, Christos Kozyrakis

faromero@stanford.edu, qianli@cs.stanford.edu, neeraja@cs.stanford.edu, kozyraki@stanford.edu

Stanford University

Abstract

Despite existing work in machine learning inference serving, *ease-of-use* and *cost efficiency* remain challenges at large scales. Developers must manually search through thousands of *model-variants* – versions of already-trained models that differ in hardware, resource footprints, latencies, costs, and accuracies – to meet the diverse application requirements. Since requirements, query load, and applications themselves evolve over time, these decisions need to be made dynamically for *each* inference query to avoid excessive costs through naive autoscaling. To avoid navigating through the large and complex trade-off space of model-variants, developers often fix a variant across queries, and replicate it when load increases. However, given the diversity across variants and hardware platforms in the cloud, a lack of understanding of the trade-off space can incur significant costs to developers.

This paper introduces INFaaS, an automated *model-less* system for distributed inference serving, where developers simply specify the performance and accuracy requirements for their applications without needing to specify a specific model-variant for each query. INFaaS generates model-variants from already trained models, and efficiently navigates the large trade-off space of model-variants on behalf of developers to meet application-specific objectives: (a) for each query, it selects a model, hardware architecture, and model optimizations, (b) it combines VM-level horizontal autoscaling with model-level autoscaling, where multiple, different model-variants are used to serve queries within each machine. By leveraging diverse variants and sharing hardware resources across models, INFaaS achieves 1.3× higher throughput, violates latency objectives 1.6× less often, and saves up to 21.6× in cost (8.5× on average) compared to state-of-the-art inference serving systems on AWS EC2.

1 Introduction

The number of applications relying on inference from Machine Learning (ML) models is already large [14,47,48,60,67] and expected to keep growing. Facebook, for instance, serves tens-of-trillions of inference queries per day [40,43]. Distributed inference dominates ML production costs: on AWS, it accounts for over 90% of ML infrastructure cost [16].

Typically, an ML lifecycle has two distinct phases – training and inference. Models are trained in the training phase; the training phase is usually characterized by long-running hyperparameter searches, dedicated hardware resource usage,

Application	Accuracy	Latency	Cost
Social Media	High	Medium	Low
Visual Guidance	High	Low	High
Intruder Detection	Low	Low	Low

Table 1: Applications querying a face recognition model with diverse requirements.

and no completion deadlines. In the inference phase, trained models are queried by various end-user applications. Being user-facing, inference serving requires cost-effective systems that render predictions with latency constraints while handling unpredictable and bursty request arrivals.

Inference serving systems face a number of challenges [61, 73] due to the following factors.

(a) Diverse application requirements: Applications issue queries that differ in latency, cost, accuracy, and even privacy [56] requirements [42,45,61]. Table 1 shows the same face recognition model queried by multiple applications with different requirements. Some applications, such as intruder detection, require inference in realtime but can tolerate lower accuracy. Other applications, such as tagging faces on social media, may prefer accuracy over latency.

(b) Heterogeneous execution environments: Leveraging heterogeneous hardware resources (e.g., different generations of CPUs, GPUs, and accelerators like TPU [49] or AWS Inferentia [18]) helps meet the diverse needs of applications and the dynamic changes in the workload; however, it is non-trivial to manage and scale heterogeneous resources [40].

(c) Diverse model-variants: Graph optimizers, such as TVM [22], TensorRT [3], and methods, such as layer fusion or quantization [15], produce versions of the same model, *model-variants*, that may differ in inference latency, memory footprint, and accuracy.

Together, these factors create a large search space. For instance, from 21 already-trained image classification models, we generated 166 model-variants, by (i) applying model graph optimizers, such as TensorRT [3], (ii) optimizing for different batch sizes, and (iii) changing underlying hardware resources (e.g., CPUs, GPUs, and Inferentia). These variants vary across many dimensions: the accuracies range from 56.6% to 82.5% (1.46×), the model loading latencies range from 590ms to 11s (18.7×), and the inference latencies for a single query range from 1.5ms to 5.7s (3,700×). Their computational requirements range from 0.48 to 24 GFLOPS (50×) [61], and the cost of hardware these variants incur [17] ranges from \$0.096/hr for 2 vCPUs to \$3.06/hr for a V100 GPU (32×). As new inference accelerators are introduced and new opti-

*Equal contribution

mization techniques emerge, the number of model-variants will only grow.

This large search space makes it hard for developers to manually map the requirements of each inference query to decisions about selecting the right model and model optimizations, suitable hardware platforms, and auto-scaling configurations. The decision complexity is further exacerbated when the load varies, applications evolve, and the availability of hardware resources (GPUs, ASICs) changes. Unlike long-running batch data analytics or ML training jobs [8, 26, 36, 55, 68] that can be right-sized during or across subsequent executions, the dynamic nature of distributed inference serving makes it infeasible to select model-variants statically.

Our key insight is that the large diversity of model-variants is not a nuisance but an opportunity: it allows us to meet the diverse and varying performance, cost, and accuracy requirements of applications, in the face of varying load and hardware resource availability, if only we can select and deploy the right model-variant effectively for *each* query. However, given the complexity of this search space, existing systems, including Clipper [25], TensorFlow Serving [5], AWS SageMaker [11], and others [1, 6, 35, 38, 75], ignore the opportunity. These systems require developers to select model-variants, batch sizes, instance/hardware types, and statically-defined autoscaling configurations, for meeting application requirements. If these decisions are made without understanding the trade-offs offered by the variants, the impact could be significant (note the wide cost, performance, and accuracy ranges spanned by the variants). We argue that in addition to traditional autoscaling, distributed inference serving systems should navigate this search space of model-variants on behalf of developers, and automatically manage model-variants and heterogeneous resources. Surprisingly, as also noted in prior work [73], no existing inference serving system does that.

To this end, we built INFaaS, an automated *model-less* system for distributed inference serving. INFaaS introduces a *model-less* interface where after registering trained models, developers specify only the high-level performance, cost, or accuracy requirements for each inference query. INFaaS generates model-variants of the registered models, and navigates the large space to select a model-variant and automatically switch between differently optimized variants to best meet the query requirements. INFaaS also automates resource provisioning for model-variants and schedules queries across a heterogeneous cluster.

To realize this, INFaaS generates model-variants and their performance-cost profiles on different hardware platforms. INFaaS tracks the dynamic status of variants (e.g., overloaded or interfered) using a state machine, to efficiently select the right variant for each query to meet the application requirements. Finally, INFaaS combines VM-level (horizontal scaling) and *model-level autoscaling* to dynamically react to the changing application requirements and request patterns. Given the large and complex search space of model-

variants, we formulate an integer linear program (ILP) for our model-level autoscaling that finds the most cost-effective combination of model-variants, to meet the goals for queries in large scale inference serving.

Using query patterns derived from real-world applications and traces, we evaluate INFaaS against existing inference serving systems, including Clipper [25] and SageMaker [11], with 175 variants generated from 22 model architectures, on AWS. Compared to Clipper, INFaaS' ability to select suitable model-variants, leverage heterogeneous hardware (CPU, GPU, Inferentia), and share hardware resources across models and applications enables it to save $1.23\times$ in cost, violate latency objectives $1.6\times$ less often, and improve resource utilization by $2.8\times$. At low load, INFaaS saves cost by $21.6\times$ compared to Clipper, and $21.3\times$ compared to SageMaker.

2 Challenges

2.1 Selecting the right model-variant

A *model-variant* is a version of a model defined by the following aspects: (a) model architecture (e.g., ResNet50, VGG16), (b) programming framework, (e.g., TensorFlow, PyTorch, Caffe2, MXNet), (c) model graph optimizers (e.g., TensorRT, Neuron, TVM, XLA [72]), (d) hyperparameters (e.g., optimizing for batch size of 1, 4, 8, or 16), and (e) hardware platforms (e.g., Haswell or Skylake CPUs, V100 or T4 GPUs, FPGA, and accelerators, such as Inferentia [18], TPU [49], Catapult [32], NPU [7]). Based on the 21 image classification models and the available hardware on AWS EC2 [9], we estimate the total number of possible model-variants would be **4,032**. The performance, cost, and accuracy trade-off space offered by these variants is large [19, 61]. As new inference accelerators are introduced and new optimization techniques emerge, the number of model-variants will only grow.

Existing inference serving systems require developers to identify the model-variant that can meet diverse performance, accuracy, and cost requirements of applications. However, generating and leveraging these variants requires a substantial understanding of the frameworks, model graph optimizers, and characteristics of hardware architectures, thus limiting the variants an application developer can leverage. As shown in Table 1, one can use the same face recognition model for several applications, but selecting the appropriate model-variant depends on the requirements of an application [61].

We argue that inference serving systems should automatically and efficiently select a model-variant for each query on behalf of developers to align with application requirements.

2.2 Reducing cost as load varies

Query patterns and service level objectives (SLOs) of applications, such as real-time translation and video analytics, can vary unpredictably [43, 50, 76]. Provisioning for peak demand leads to high cost, hence distributed inference serving systems need to dynamically respond to changes. Traditional autoscaling focuses on horizontal virtual machine replication

Variant (hardware, framework)	Lat. (ms)	Req/s	Cost (\$/s)
A (4 CPUs, TensorFlow)	200	5	1
B (1 Inferentia core, Neuron)	20	100	3
C (1 V100 GPU, TensorRT)	15	800	16

Table 2: Latency, saturation throughput, and normalized cost (based on AWS pricing) for three ResNet50 variants.

QPS	SLO (ms)	#Var. A	#Var. B	#Var. C	Cost (\$/s)
10	300	2	0	0	2
10	50	0	1	0	3
1000	300	0	2	1	22

Table 3: Cheapest configuration (in #instances) of variants from Table 2 to meet the QPS and SLO; last column shows total cost.

(VM-scaling), adding or removing worker machines [33, 37]. However, relying only on worker replication may incur significant latency, as new machines must be spawned.

Autoscalers used by existing inference serving systems [11, 35, 37] replicate a statically-fixed (developer-specified) model-variant for all the queries of an application. This is insufficient because: (a) the right variant may change with load (e.g., a CPU variant may be more suitable at low QPS to meet the cost SLOs) and (b) the hardware resources needed to replicate the same variant may not always be available (e.g., shortage of GPU instances at some point).

In addition to using VM-scaling and replication-based model-scaling, we introduce *model-level vertical scaling*, where we switch to a differently optimized variant as load changes. The challenge is to identify which variant to scale to given available hardware resources and query requirements. Consider the example shown in Table 2 with three ResNet50 variants. Each variant runs on a different hardware resource and differs in latency, saturation throughput, and cost. In Table 3, we present three input loads and SLO requirements, with the goal of scaling to the most cost-effective combination of variants. In the first case (QPS = 10 and SLO = 300ms), though all variants can meet the QPS and SLO, using two instances of Variant A is the cheapest choice (8× cheaper than using Variant C). In the second case, the load remains unchanged, but due to the stricter SLO, Variant B becomes the cheapest choice (5.3× cheaper than using Variant C). In the third case, the combination of two instances of Variant B and one of Variant C is the cheapest. This configuration is 9× cheaper than using 200 instances of Variant A (the most expensive configuration). Deciding the best configuration becomes more challenging as the number of variants increases and resource availability changes.

2.3 Improving utilization at low load

For predictable performance, one may serve each model-variant on a dedicated machine to exclusively access hardware resources. But, this often results in underutilized resources and cost-inefficiency, especially at low load. Instead, the serving systems should support multi-tenancy by *sharing resources*

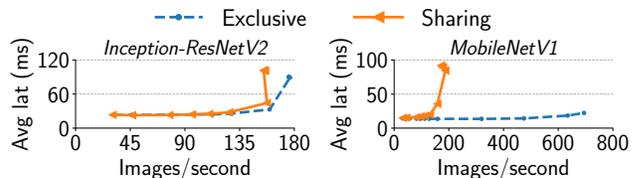


Figure 1: Impact of co-locating Inception-ResNetV2 and MobileNetV1 on a V100 GPU. Both variants are TensorRT, batch-1, FP16. Graphs show average latency and throughput for each model running alone vs sharing, subjected to the same (QPS).

across applications and models, thereby improving utilization and the overall cost. Recent work [36, 71] has shown the benefits of sharing GPUs for deep-learning training jobs. ML inference jobs are typically less demanding for compute and memory resources than the training jobs, making inference jobs ideal for sharing GPUs and other accelerators [46, 74].

However, how to share accelerators across multiple tenants while maintaining predictable performance is not obvious. Figure 1 shows the result of co-locating a large (Inception-ResNetV2) and a small (MobileNetV1) model on a GPU. At low load, sharing a GPU does not affect the performance of either model. At higher load, this co-location heavily impacts the performance of the small model, while the large model remains unaffected. The point when co-location starts affecting the performance varies across models, and depends on both the load and the hardware architecture.

3 INFaaS

Design principles. We design INFaaS based on the following guidelines. First, INFaaS should support a declarative API: Developers should not need to specify the model, model optimizations, suitable hardware platforms, or autoscaling configurations; they should only focus on high-level performance, cost, or accuracy requirements. Second, INFaaS should automatically and efficiently select a model-variant, while considering the dynamic state of the model-variants and the hardware resources, for (a) serving each query, and (b) deciding how to scale in reaction to changing application load. Third, to improve resource utilization, the system should share hardware resources across model-variants and applications, without violating performance-cost constraints. Finally, the system design should be modular and extensible to allow new model-variant selection policies. By following these design principles, we naturally address the challenges raised in Section 2.

Functionality. INFaaS generates new model-variants from the models registered by developers, and stores them in a repository (Section 3.2). These variants are optimized along different dimensions using model graph optimizers such as Neuron and TensorRT. For each inference query, INFaaS automatically selects a model-variant to satisfy its performance, cost, and accuracy objectives (Section 4.1). INFaaS’ autoscaler combines VM-level autoscaling with model-level horizontal and vertical autoscaling to meet application per-

API	Parameters
register_model	modelName, modelBinary, valSet, appID
inference_query	input(s), appID, latency, accuracy
inference_query	input(s), modelName

Table 4: INFaaS’ declarative developer API.

formance and cost requirements while improving utilization of resources (Section 4.2). INFaaS introduces *model-vertical autoscaling* that, through model selection, upgrades or downgrades to a differently optimized model-variant by leveraging the diversity of model-variants (Section 4.2.1). INFaaS efficiently maintains static and dynamic profiles of model-variants and hardware resources to support low latencies for selecting and scaling model-variants (Sections 3.2 and 4). Finally, INFaaS features the model-variant selection policy described in Section 4, but allows developers to extend and customize it.

3.1 Model-less interface for inference

Table 4 lists INFaaS’ model-less API.

Model registration. Developers register one or more models using the `register_model` API. This API accepts a developer-assigned model identifier (`modelName`), the model (`modelBinary`) in serialized format (e.g., a TensorFlow SavedModel or model in ONNX format), and a developer-assigned application identifier (`appID`). Models for different prediction tasks within the same application (e.g., optical character recognition and language translation) can be registered with separate `appIDs`. Lines 1-2 in Figure 2 show how a developer registers two models, a ResNet50 and a MobileNet, for an application with `appID=detectFaceApp`. INFaaS generates multiple variants from these already trained models. For instance, using ResNet50 alone, INFaaS can generate about 50 variants by changing the batch size, the hardware, and the model graph optimizer (Section 3.2). Note that INFaaS is an inference serving system and does not train new models; INFaaS only generates variants from already-trained models. The `register_model` API takes a validation dataset (e.g., `valSet`) as input to calculate the accuracy of the newly generated variants. For each incoming query, INFaaS automatically selects the right model-variant to meet the specified goals.

Query submission. Being declarative, INFaaS’ API allows developers to specify high-level goals without needing to specify the variants for their queries. Using the `inference_query` API, developers can submit inference queries in two ways:

- *Specifying application requirements.* Developers may submit queries for their application and specify high-level application performance, cost, and accuracy requirements (e.g., Line 3 in Figure 2). INFaaS then navigates the search space of model-variants for the given application, and selects model-variants and scaling strategies. For instance, for a query with `appID=detectFaceApp`, INFaaS searches for a suitable variant of ResNet50 and MobileNet to meet the goal of latency (200ms) and accuracy (above 70%).

```

1 register_model("ResNet50", ResNet50.pt, valSet, detectFaceApp)
2 register_model("MobileNet", MobileNet.pt, valSet, detectFaceApp)
# Developer registered 2 models for the detectFaceApp;
# INFaaS generates variants from these two registered models
3 inference_query(input.jpg, detectFaceApp, 200ms, 70%)
# Developer submitted a query with "input.jpg" as the input
# and specified requirements; INFaaS then selects a variant
# automatically to meet 200ms latency and accuracy > 70%

```

Figure 2: Registering models and submitting queries with INFaaS.

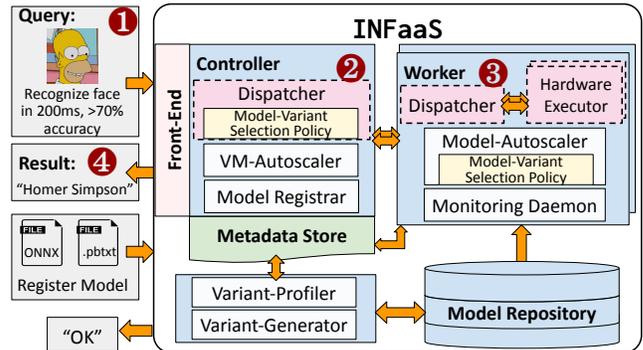


Figure 3: Architecture of INFaaS. Modules in boxes with dashed border are on the critical path of serving queries. All other modules do not impact serving latency. Numbered circles correspond to the typical life-cycle of queries.

- *Specifying a registered model.* Developers may use this interface to specify the model, `modelName`, they registered for the corresponding application (e.g., "ResNet50" for `detectFaceApp`). This interface supports developers who want direct control over the model-variant used. This is the only option offered by existing inference systems. INFaaS then dynamically scales resources for the specified model-variant based on the observed workload.

INFaaS’ serving workflow for an inference query. Applications interact with INFaaS by submitting inference queries through the *Front-End*, logically hosted at the *Controller* (see steps marked in Figure 3). The Controller selects a model-variant and dispatches the inference query to a *Worker* machine. Workers further send inference queries to the appropriate *Hardware Executors* according to the selected model-variant, and reply with inference results to applications.

3.2 Architecture

We now describe INFaaS’ architecture (Figure 3) in detail.

Controller. The logically-centralized controller receives model registration and inference requests. The controller hosts three modules: (a) The Dispatcher that uses the model-variant selection policy for selecting a variant to serve a query, (b) The VM-Autoscaler that is responsible for scaling the number of workers up and down based on the current load and resource utilization, and (c) The Model Registrar that handles model registration.

Workers. Worker machines execute inference queries assigned by the controller. Hardware-specific Executor daemons manage the deployment and execution of model-variants. The

Dispatcher forwards each query to a specific model-variant instance through the corresponding hardware executor. The Model-Autoscaler detects changes in the load and decides a scaling strategy that either replicates running variants or selects a different variant, within the worker. It uses the model-variant selection policy to select a variant to scale to. The Monitoring Daemon tracks the utilization of machines and variants, and manages resources shared by multiple variants to avoid SLO violations.

Variant-Generator and Variant-Profiler. From the registered models, the Variant-Generator generates model-variants optimized for different batch sizes, hardware, and hardware-specific parameters (e.g., number of cores on Inferentia) using model graph optimizers, including TensorRT [3] and Neuron [15]. INFaaS uses the validation set submitted by the developer to calculate the accuracy of the newly generated variants; INFaaS records this information in the Metadata Store. The Variant-Generator does not train or produce new model architectures: variants are generated only from registered models. To help model-variant selection, the Variant-Profiler conducts one-time profiling for each variant where it measures statistics, such as the loading and inference latencies, and peak memory utilization. These parameters, along with the corresponding `appID`, accuracy, and maximum supported batch size are recorded in the Metadata Store. After profiling, a variant is saved in the Model Repository. The total profiling time for all generated variants from a submitted model is a few minutes on a single VM with the variant’s target hardware. This profiling cost will be amortized over long-term serving time in production settings.

Model-Variant Selection Policy. INFaaS invokes the model-variant selection policy in two cases.

(Case I) On arrival of a query: The controller’s Dispatcher uses the policy to select a variant for each incoming query. This model-variant selection lies on the critical path of serving each query. To reduce the latency of decision-making, we designed an efficient variant search algorithm (Section 4.1).

(Case II) On changes in query load: As the query load changes, the worker’s Model-Autoscaler uses the policy to determine whether to replicate existing variants, or vertically scale to a different variant. The Model-Autoscaler monitors the incoming query load and the current throughput of INFaaS to detect the need for scaling. If a change is detected, the Model-Autoscaler invokes the policy in the background to select a suitable scaling strategy (Section 4.2).

To allow for other model-variant selection algorithms, we designed INFaaS to decouple policies from mechanisms [53].

Metadata Store. The Metadata Store enables efficient access to the static and dynamic data about workers and model-variants; this is needed for making model-variant selection and scaling decisions. This data consists of (a) the information about available model architectures and their variants (e.g., accuracy and profiled inference latency), and (b) the resource usage and load statistics of variants and worker machines. The

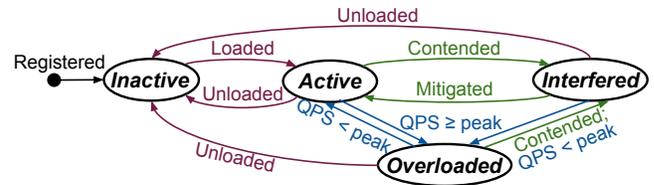


Figure 4: State machine capturing the dynamically changing status of model-variants.

Metadata Store strategically uses data structures to ensure low access latencies ($\sim O(1)$) for efficient decision-making. The Metadata Store runs on the same machine as the controller to reduce access latencies for selecting variants. Implementation and data structure details are described in Section 5.

Model Repository. The Model Repository is a high-capacity persistent storage medium that stores serialized variants that are accessible to workers when needed to serve queries.

4 Selecting and Scaling Model-Variants

INFaaS uses the model-variant selection policy in two cases: (I) On arrival of a query: INFaaS’ controller needs to select a variant for each query to meet an application’s high-level requirements (Section 4.1). This invocation of the selection policy lies on the critical path of inference serving. (II) On changes in query load: As the query load changes, INFaaS’ workers must decide whether to switch to a differently optimized variant (Section 4.2). The worker invokes the selection policy off the critical path. INFaaS provides an internal API, `getVariant`, for invoking model-variant selection policy. In both cases, INFaaS needs to consider both the static and dynamic states of variants and available resources. Only considering statically-profiled metadata is insufficient, since the following aspects can significantly impact the observed performance and cost: a selected variant (a) may not be loaded, hence we need to consider its loading latency, (b) may be already loaded but serving at its peak throughput, (c) may be already loaded but experiencing resource contention from co-located inference jobs, and (d) may not be loaded due to lack of resources required for that specific variant. We next describe how INFaaS tracks the dynamic state of model-variants, and then describe the policy used in the two cases.

State machine for the lifecycle of model-variants. To track the dynamic state of each model-variant instance per-application, INFaaS uses a state machine (shown in Figure 4). All the registered and generated model-variants start in the *Inactive* state: they are not loaded on any worker. Once a variant instance is loaded, it transitions to the *Active* state. These variant instances are serving less than their peak throughput, tracked by the worker’s monitoring daemons. Variant instances enter the *Overloaded* state when they serve at their peak throughput. Finally, variant instances in the *Interfered* state are not overloaded but are still experiencing higher inference latencies than the profiled values. Interference occurs when co-located variants contend over shared resources (e.g., caches, memory bandwidth, or hardware threads).

Algorithm 1 Model-variant selection for case I (arrival of a query)

```
1: function GETVARIANT(appID, accuracy, latency)
2:   if searchActiveVariants(appID, accuracy, latency) then
3:     Get least-loaded worker,  $W_{1l}$ , running activeVariant
4:     return activeVariant,  $W_{1l}$ 
5:   if searchInactiveVariants(appID, accuracy, latency) then
6:     Get lowest-util worker,  $W_{1u}$ , with inactiveVariant's HW
7:     return inactiveVariant,  $W_{1u}$ 
8:   return suggestVariant(appID, accuracy, latency)
```

Maintaining the state machine. Each model-variant instance's state machine is maintained by the worker's monitoring daemons and is organized in the Metadata Store for fast access. This enables INFaaS' Dispatcher to use the model-variant selection policy for serving queries on the order of hundreds of μ s to ms (assessed further in Section 6.4). State machine implementation details are described in Section 5.

4.1 Case I: On arrival of a query

When a query arrives, INFaaS' Dispatcher invokes the `getVariant` method of model-variant selection policy to choose a variant (Algorithm 1). For this case, the input to `getVariant` is the query's requirements, and the output is the variant and worker to serve the query. `getVariant` first checks whether any variants in the *Active* state match the query's requirements (Line 2). Variants in *Active* state do not incur a loading latency. If such a variant is found, INFaaS dispatches the query to the least-loaded worker running the variant instance (Lines 3-4). Otherwise, INFaaS considers variants in the *Inactive* state: `getVariant` first enquires the Metadata Store and retrieves the variant with the lowest combined loading and inference latency that matches the query's requirements (Line 5). If such a variant is found, INFaaS sends the query to the worker with the lowest utilization on the variant's target hardware (Lines 6-7). Otherwise, since no registered or generated variant can meet the developer's requirements, INFaaS suggests a variant that can achieve the closest target accuracy and/or latency (Line 8). We assess the efficiency of this policy over brute-force search in Section 6.4.

Mitigating performance degradation. For better resource utilization, INFaaS co-locates variants on hardware resources; as a result, they may interfere and cause SLO violations. To prevent such violations, INFaaS avoids selecting variants that are in the *Interfered* or *Overloaded* state. For interfered variants, INFaaS triggers a mitigation process in the background to avoid affecting the performance of online serving. If there are idle resources available on the same worker, INFaaS migrates the variant in the *Interfered* state to the available resources (e.g., a different set of cores). This avoids the need to fetch a variant from the model repository. If no resources are available for loading the variant on the worker, the worker asks the controller's Dispatcher to place the variant on the least-loaded worker. For variants in the *Overloaded* state, INFaaS' Model-Autoscaler assesses whether it is more cost-effective to scale to a different variant (see Section 4.2.1).

Extensibility. The state machine and model-variant selection policy are extensible. For instance, `getVariant` can be extended to prioritize particular variants in the *Active* state (e.g., prefer least power-hungry variants).

4.2 Case II: On changes in query load

As query load changes, INFaaS needs to revisit its variant selection decision to check whether a different variant is more cost-efficient. Existing inference serving systems [5, 11, 25, 35, 37] are agnostic to the diversity of model-variants, and only replicate a statically fixed (developer-specified) model-variant for all the queries of an application. However, as discussed in Section 2.2, autoscaling that replicates the same model-variant alone is not enough because: (a) the right model-variant changes with load and (b) the required resources might not be available to replicate a specific variant.

For INFaaS' autoscaling, in addition to using traditional VM-level, horizontal autoscaling, we introduce *model-vertical scaling*: change (upgrade or downgrade) to a differently optimized model-variant, thus leveraging the diversity of model-variants. INFaaS' autoscaling is a joint effort between the workers and the controller. Each worker hosts a Model-Autoscaler that consults with the model-variant selection policy to make model-level autoscaling decisions (Sections 4.2.1, and 4.2.2). The controller hosts a VM-Autoscaler that makes VM-level autoscaling decisions (Section 4.2.3).

4.2.1 Model-Autoscaler at each worker

To react to the changes in query load, INFaaS' Model-Autoscaler needs to decide the type and number of model-variants to use while minimizing the cost of running the variants. To figure out the type and number of model-variants needed, we formulated the following integer linear program (ILP) that decides a scaling action (replicate, upgrade, or downgrade) for each variant. This ILP minimizes the total cost of scaling actions for all the variants to meet the incoming query load.

Formulation. For an application, the outcome (optimization variable) of our ILP is the optimal scaling action, δ_{ij} , for each model-variant v_{ij} , variant j of model architecture i . δ_{ij} is an integer that captures the scaling action as follows: (a) A positive value denotes loading instances of the variant, (b) a negative value denotes unloading instances of this variant, and (c) a value of zero denotes no scaling needed. For a variant v_{ij} that is already loaded, a positive value of δ_{ij} indicates a *replicate* action. A positive value of δ_{ij} for a variant v_{ij} that is not already loaded indicates an *upgrade* or *downgrade* action depending on the hardware cost of v_{ij} .

The objective function that our ILP minimizes is the total cost of all the chosen scaling actions. For a variant v_{ij} , this cost for an action δ_{ij} is the sum of the hardware cost (in \$/second), and the loading latency (in seconds) of the variant:

$$\text{Cost}(\delta_{ij}) = C_{ij}(\delta_{ij} + \lambda T_{ij}^{\text{load}} \max(\delta_{ij}, 0))$$

where C_{ij} is the hardware cost (in \$/second) for running the variant, T_{ij}^{load} is the loading latency of the variant, and λ (in

$\frac{1}{\text{second}}$) is a tunable parameter for the query load unpredictability. Large values of λ place more weight on minimizing loading latency to meet SLOs when the query load is unpredictable or spiky. Small values of λ place more weight on minimizing the hardware cost when the query load is more stable.

Thus, our objective function representing the total cost for all the variants is: $\sum_{i,j} \text{Cost}(\delta_{ij})$. We impose the following constraints on our ILP:

- (1) With the chosen scaling actions, INFaaS supports the incoming query load.
- (2) The newly-loaded instances satisfy applications' SLOs.
- (3) The resources consumed by all variants do not exceed the total system resources.
- (4) The number of running instances is non-negative.

We write these constraints formally as:

$$\begin{aligned} \sum_{i,j} Q_{ij}(N_{ij} + \delta_{ij}) &\geq L + \text{slack} && \text{for all } i, j && (1) \\ T_{ij}^{\text{inf}} &\leq S && \text{if } \delta_{ij} > 0 && (2) \\ \sum_{i,j} R_{ij}^{\text{type}}(N_{ij} + \delta_{ij}) &\leq R_{\text{total}}^{\text{type}} && \text{for all types} && (3) \\ N_{ij} + \delta_{ij} &\geq 0 && \text{for all } i, j && (4) \end{aligned}$$

where (a) Q_{ij} : the saturation QPS of variant v_{ij} , (b) N_{ij} : the number of running instances of variant v_{ij} , (c) L : the incoming query load, (d) slack : configurable headroom to absorb sudden load spikes, (e) T_{ij}^{inf} : the inference latency of variant v_{ij} , (f) T_{ij}^{load} : the loading latency of variant v_{ij} , (g) S : SLO of the considered application, (h) R_{ij}^{type} : the resource requirements of variant v_{ij} , for a resource type (CPU cores, CPU memory, GPU memory, number of Inferentia cores), and (i) $R_{\text{total}}^{\text{type}}$: the total available amount of resources of a type (CPUs, GPUs, Inferentia cores) on the underlying worker machine.

The model-variant selection policy queries the Metadata Store to get the values of these variables.

Practical limitation of the ILP. Unfortunately, this ILP is NP-complete and hence offers limited practical benefits [34, 54, 69]: it has to exhaustively search through all the model-variants, track their dynamically changing state, and accurately estimate the QPS each variant can support to find a scaling configuration that can sustain the changed query workload. Gurobi [41] took 23 seconds to find the optimal number of running variant instances across 50 model architectures, and 50 seconds for 100 model architectures. To meet realtime requirements of latency-sensitive applications, INFaaS must have sub-second response time to query workload changes.

4.2.2 A Greedy Heuristic

The time taken to solve each instance of our ILP makes it impractical to use for INFaaS. Instead, we design a greedy heuristic algorithm that replaces our ILP's large search space by a subset of model-variants. This pruned search space allows INFaaS to meet the outlined constraints at sub-second latency. We evaluate the effectiveness of this algorithm in Section 6.2. Each worker machine runs a Model-Autoscaler that together with the model-variant selection policy approximates

this ILP as follows: (a) Identify whether the constraints are in danger of being violated, (b) Consider two strategies, replicate or upgrade/downgrade, to satisfy the constraints, (c) Compute the objective for each of these scaling actions and pick the one that minimizes the objective cost function, and (d) Coordinate with the controller to invoke VM-level autoscaling if constraints cannot be satisfied with model-level autoscaling.

Scaling up algorithm: To decide if there is a need to scale (Constraint #1), the Model-Autoscaler estimates the current headroom in capacities of running model-variants, given the profiled values of their saturation throughput, and the current load they are serving. We compute the current load served by a variant using the batch size and number of queries served per second. The load served by a worker is estimated by summing the load served by all running variants. The saturation throughput of all running variants is estimated in a similar manner using the profiled values of model-variants. The Model-Autoscaler then computes the current headroom of a worker as the ratio of the combined saturation throughput and the combined load currently served by the running variants on that worker. INFaaS maintains a minimum headroom, `slack-threshold`, on each worker to absorb sudden load spikes. We discuss the value of this tunable parameter in Section 5. When the current headroom is below the required minimum `slack-threshold`, the Model-Autoscaler concludes that we need to scale, and proceeds to answer the second question: *how* to scale (replicate or upgrade) to meet the incoming query load.

To decide how to scale, the Model-Autoscaler uses the model-variant selection policy's `getVariant` method to select the cheapest option between replication and upgrading. For this case, the input to `getVariant` is the incoming query load, and the output is the set of scaling actions. The policy first estimates the cost of model-horizontal scaling (replication) by estimating the number of instances of the running variant that would be added to meet the incoming query load (Constraints #1, #4). Secondly, the policy estimates the cost of model-vertical scaling (upgrade), by querying the Metadata Store to select variants of the same model architecture that can meet the SLO (Constraint #2), and support a higher throughput than the currently running variant. The required number of instances for these variants to meet the incoming query load is then estimated. Finally, the model-variant selection policy computes the cost function of our ILP, by using the hardware cost (\$/s) and the variant loading latency to decide whether to replicate the running variant, or upgrade to a variant that supports higher throughput. The available resources on the worker limit the number of variant instances it can run (Constraint #3). Thus, if the strategy requires more resources than are available on the current worker (e.g., hardware accelerator), the worker coordinates with the controller to load the variant on a capable worker.

Scaling down algorithm: To decide if and how to scale down (remove replicas or downgrade), the Model-Autoscaler

on each worker uses the model-variant selection policy that follows a similar algorithm explained above for scaling up. At regular intervals, this policy checks if the incoming query load can be supported by removing an instance of the running variant, or downgrading to a cheaper variant (optimized for a lower batch size or running on different hardware). The Model-Autoscaler waits for T_v time slots before executing the chosen strategy for a variant v , to avoid scaling down too quickly. T_v is set equal to the loading latency of variant v .

Comparison with ILP. As described in Section 4.2.1, the ILP does not have sub-second response time. Setting a larger headroom to allow the ILP to produce a solution can result in (a) scaling variants too quickly, which leads to underutilization and higher cost, and (b) violating SLOs during unpredictable load spikes. Besides traditional model-horizontal scaling, our model-vertical scaling further reduces cost by upgrading to a variant that supports higher throughput. Thus, INFaaS matches the throughput of the optimal solution, while the deviance from the ILP is bounded by the difference between the optimal cost and the cost of replicating running variants. Our greedy heuristic reacts to load changes (e.g., load spikes) within sub-second response time while reducing the cost over multiple scaling actions.

4.2.3 VM-Autoscaler at controller

In addition to model-level scaling, INFaaS also scales the worker machines for deploying variants. Following the mechanisms used in existing systems [11, 20, 25, 35, 44], the VM-Autoscaler decides when to bring a worker up/down:

1. When the utilization of any hardware resource exceeds a configurable threshold across all workers, the VM-Autoscaler adds a new worker with the corresponding hardware resource. We empirically set the threshold to 80%, considering the time to instantiate VMs (20-30 seconds): a lower threshold triggers scaling too quickly and unnecessarily adds workers; a higher value may not scale in time.
2. When variants on a particular hardware platform (e.g., GPU) are in the *Interfered* state across all workers, the VM-Autoscaler adds a worker with that hardware resource.
3. When more than 80% of workers have *Overloaded* variants, the VM-Autoscaler starts a new worker.

To improve utilization, INFaaS dispatches requests to workers using an online bin packing algorithm [64].

5 Implementation

We implemented INFaaS in about 20K lines of C++ code¹. INFaaS' API and communication logic between controller and workers are implemented using gRPC in C++ [2]. Developers can interact with INFaaS by issuing gRPC requests in languages supported by gRPC, such as Python, Go, and Java. INFaaS uses AWS S3 [10] for its Model Repository. The model-variant selection policy is implemented as an extensible C++ library that is linked into the controller's Dis-

patcher and worker's Model-Autoscaler. `getVariant` is a virtual method, and can be overridden to add new algorithms.

On the controller machine, the Front-End, Dispatcher, and Model Registrar are threads of the same process for efficient query dispatch. The Dispatcher is multi-threaded to support higher query traffic. The VM-Autoscaler is a separate process, that polls system status periodically. We swept the polling interval between 0.5-5 seconds at 0.5 second increments (similar to prior work [51, 63]), and arrived at a 2 seconds polling interval. Longer intervals did not scale up fast enough, especially during load spikes, and shorter intervals were too frequent given VM start-up latencies.

On worker machines, the Dispatcher and monitoring daemon run as separate processes. Every 2 seconds, the monitoring daemon updates compute and memory utilization of the worker, loading, and average inference latencies, along with the current state (as noted in Figure 4) for each variant running on that worker, to the Metadata Store. We deployed custom Docker containers for PyTorch and Inferentia variants, and leveraged Triton Inference Server-19.03 [6] to support TensorRT, Caffe2, and TensorFlow variants on GPU. We used the TensorFlow Serving container for TensorFlow variants on CPU [5]. The Model-Autoscaler's main thread makes scaling decisions periodically. We swept the same range (0.5-5 seconds at 0.5 second increments) as the VM-Autoscaler, and arrived at a 1 second polling interval. The interval is shorter than the VM-Autoscaler's polling interval as model loading latencies are shorter than VM start-up latencies. The main thread also manages a thread pool for asynchronously loading and unloading model-variants. To tune `slack-threshold`, we explored values between 1.01 and 1.1 [31], and set it to 1.05. In our setup, lower thresholds did not scale variants fast enough to meet load changes, while higher thresholds scaled variants too quickly.

We built the Variant-Generator using TensorRT [3] and Neuron [15]; it is extensible to other similar frameworks [22, 59]. For each variant, the Variant-Profiler records the latency for batch sizes from 1 to 64 (power of two increments). For natural language processing models, we record the latencies of varying sentence lengths for each of these batch sizes.

We built the Metadata Store using Redis [62] and the Redox C++ library [4]. The Metadata Store uses hash maps and sorted sets for fast metadata lookups that constitute the majority of its queries. Per-application, each model-variant instance's state is encoded as a {variant, worker} pair that can be efficiently queried by the controller and worker.

6 Evaluation

We first compare INFaaS with all of its optimizations and features to existing systems (Section 6.1). To further demonstrate the effectiveness of INFaaS' design decisions and optimizations, we evaluate its individual aspects: model-variant selection, scaling (Section 6.2), and SLO-aware resource sharing (Section 6.3). Finally, we quantify the overheads of INFaaS'

¹<https://github.com/stanford-mast/INFaaS>

Features	Clipper, TFS, TIS	SageMaker, AI Platform	INFaaS
Model-less abstraction	No	No	Yes
Variant selection	Static	Static	Dynamic
VM-autoscaling	No	Yes	Yes
Model-horizontal scaling	No	Yes	Yes
Model-vertical scaling	No	No	Yes
SLO-aware resource sharing	No	No	Yes

Table 5: Comparison between INFaaS and the baselines.

decision-making (Section 6.4). We begin by describing the experimental setup common across all experiments, the baselines, the model-variants, and the workloads.

Experimental setup. We deployed INFaaS on a heterogeneous cluster of AWS EC2 [9] instances. We hosted the controller on an `m5.2xlarge` instance (8 vCPUs, 32GiB DRAM), and workers on `inf1.2xlarge` (8 vCPUs, 16GiB DRAM, one AWS Inferentia), `p3.2xlarge` (8 vCPUs, 61GiB DRAM, one NVIDIA V100 GPU), and `m5.2xlarge` instances. All instances feature Intel Xeon Platinum 8175M CPUs operating at 2.50GHz, Ubuntu 16.04 with 4.4.0 kernel, and up to 10Gbps networking speed.

Baselines. To the best of our knowledge, no existing system provides a model-less interface like INFaaS; state-of-the-art serving systems require developers to specify the variant and hardware. For a fair comparison, we configured INFaaS to closely resemble the resource management, autoscaling techniques, and APIs of existing systems, including TensorFlow Serving [5] (TFS), Triton Inference Server (TIS) [6], Clipper [25], AWS SageMaker [11] (SM), and Google AI Platform [35]. Specifically, we compared INFaaS to the following baseline configurations for query execution:

- **Clipper⁺:** Derived from TFS, TIS, and Clipper, this baseline pre-loads model-variants, and requires developers to set a pre-defined number of variant instances. Thus, we set the number of variant instances such that Clipper⁺ achieves the highest performance given available resources.
- **SM⁺:** Derived from SageMaker and AI Platform, this baseline scales each model-variant horizontally, but does not support model-vertical scaling that INFaaS introduces.

Table 5 lists the differences between baselines and INFaaS. Configuring the baselines with INFaaS (a) allowed for a fair comparison by removing variabilities in execution environments (e.g., RPC and container technologies), and (b) enabled us to evaluate our design decision individually by giving the baselines access to INFaaS’ features and optimizations, including: support for model graph optimizations, and INFaaS’ detection and mitigation of variant performance degradation.

Clipper vs Clipper⁺. To validate our baseline configurations through INFaaS, we evaluated Clipper⁺ against the open-source Clipper deployment (Clipper) [23] with its adaptive batching and prediction caching features enabled. We deployed two ResNet50 TensorFlow CPU instances for each. For Clipper, we swept its adaptive batching SLO from 500ms to 10 seconds, and found it achieved its maximum

Model Family (Task)	#Vars	Model Family (Task)	#Vars
MobileNet (classification)	13	VGG (classification)	30
AlexNet (classification)	9	Inception (classification)	25
DenseNet (classification)	22	NasNet (classification)	6
ResNet (classification)	61	GNMT (translation)	9

Table 6: Model architectures, tasks, and associated variants.

throughput (7 QPS) when setting the SLO to 1 second. For the same SLO, Clipper⁺ was able to achieve 10 QPS. As prior work has noted [66], Clipper’s adaptive batching is insufficient for maintaining a high QPS, because it relies on an external scheduler to allocate resources for it. Since Clipper⁺ benefits from INFaaS’ resource allocation and management, variant performance degradation detection and mitigation, and variant optimizations, we use Clipper⁺ in the place of Clipper for the remainder of our evaluation.

SageMaker vs SM⁺. We also validated that the latency and throughput of CPU, GPU, and Inferentia variants with SM⁺ closely match SageMaker, while offering the benefits outlined for Clipper⁺. Thus, we use SM⁺ in the place of SageMaker as our baseline.

Model-variants. Guided by the MLPerf Inference benchmark [61], we collected a variety of models. Table 6 shows the 8 model families (22 architectures) and the number of associated variants. Our models are pre-trained using Caffe2, TensorFlow, and PyTorch. Our classification models are pre-trained on ImageNet [30]; translation models are pre-trained on the WMT16 [70] English-German dataset. We generated 175 variants in total, differing in the frameworks (Caffe2, TensorFlow, PyTorch), compilers (TensorRT, Neuron), batch sizes (1 to 64), and hardware platforms (CPU, GPU, Inferentia).

Workloads. We evaluated using both synthetic and real-world application query patterns. For synthetic workloads, we used common patterns [29] indicating flat and fluctuating loads, with a Poisson inter-arrival rate [39, 61]. For real-world inference workloads, we used the timing information from a Twitter trace from 2018 collected over a month [13] since there is no publicly available inference serving production traces. Twitter queries are likely passed through hate speech detection models [28] before being posted. Furthermore, as noted in recent work on inference serving [75], this trace resembles real inference workloads with both diurnal patterns and unexpected spikes (consistent with production serving workloads [65]). For each experiment, we randomly selected one day out of the month from the Twitter trace. We also set the accuracy such that it can always be satisfied by the registered variants; INFaaS’ handling of infeasible accuracy requirements is discussed in Section 4.1.

6.1 INFaaS with production workload

We now show that through model selection, resource allocation, and autoscaling mechanisms, INFaaS improves the throughput, cost, utilization, and reduces SLO violations.

Experimental setup. We mapped the Twitter trace to a range between 10 and 1K QPS for a total of 113,420 batch-1 queries.

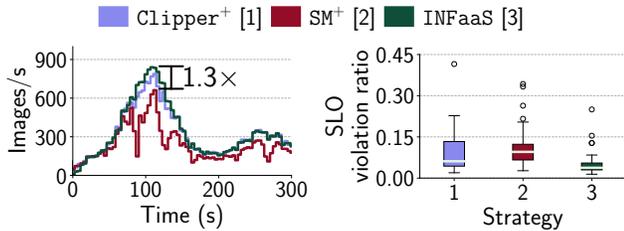


Figure 5: Throughput and SLO violation ratio (# of SLO violations by total # of queries), measured every 4 seconds. Each box shows the median, 25% and 75% quartiles; whiskers extend to the most extreme non-outlier values ($1.5\times$ interquartile range). Circles show the outliers.

We used all 22 model architectures. Based on prior work [52], we used a Zipfian distribution for model popularity. We designated 4 model architectures (DenseNet121, ResNet50, VGG16, and InceptionV3) to be *popular* with 50ms SLOs and share 80% of the load. The rest are *cold* models with SLO set to $1.5\times$ the profiled latency of each model’s fastest CPU variant. Baselines statically selected GPU variants for popular models, and CPU variants for the rest; they used 5 CPU and 7 GPU workers. INFaaS started with 5 CPU, 5 GPU, and 2 Inferentia workers. We computed the worker costs based on AWS EC2 pricing [17].

Results and discussion. Figure 5 shows INFaaS achieved $1.1\times$ and $1.3\times$ higher throughput, and $1.63\times$ and $2.54\times$ fewer SLO violations compared to Clipper⁺ and SM⁺, respectively. INFaaS scaled models both horizontally and vertically: it upgraded to Inferentia or GPU (higher batch) variants when needed. In reaction to the increased load, INFaaS added a 3rd Inferentia worker at 40 seconds. Although SM⁺ scales variants horizontally, it achieved lower throughput and violated more SLOs due to frequently incurring variant loading penalties and being unable to upgrade variants. By leveraging variants that span heterogeneous hardware (CPU, GPU, Inferentia), INFaaS achieved $1.23\times$ lower cost, while keeping SLO violations under 4% on average. INFaaS also load-balanced requests and mitigated overloaded or interfered variants. This resulted in an average worker utilization of 48.9%, with an average GPU DRAM utilization of 58.6%. The latter is $5.6\times$ and $2.8\times$ higher than SM⁺ and Clipper⁺, respectively.

INFaaS achieved higher performance ($1.3\times$ higher throughput) and resource utilization ($5.6\times$ higher GPU utilization), and lower SLO violations ($2\times$ lower) and cost ($1.23\times$ lower) compared to the baselines.

6.2 Selecting and scaling model-variants

Next, we show the efficiency of INFaaS’ model-variant selection policy to select and vertically scale the variants.

Experimental setup. To compare INFaaS with common configurations developers would choose today, we considered two cases for a model: only GPU variants are used (Clipper⁺_{GPU}, SM⁺_{GPU}) and only CPU variants are used (Clipper⁺_{CPU}, SM⁺_{CPU}). We used variants derived from one model architecture, ResNet50, and one worker. Clipper⁺_{CPU}

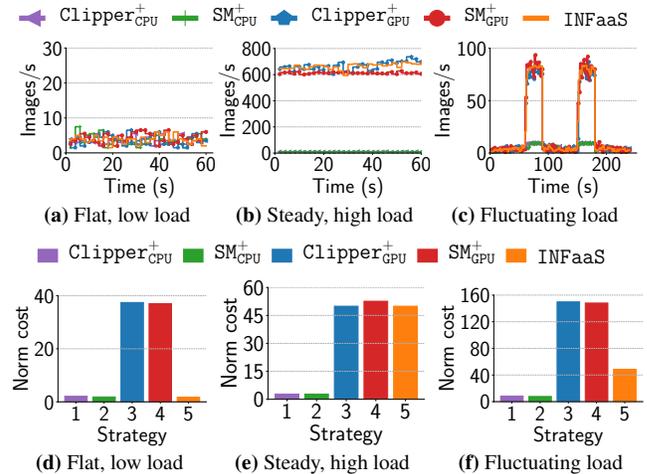


Figure 6: Throughput (top) and cost (bottom), with ResNet50 and batch-1 requests. INFaaS reduced cost and met the load.

pre-loads and persists 2 instances of the TensorFlow CPU variant. Clipper⁺_{GPU} persists one TensorRT variant optimized for batch size of 8, configured to serve the provided peak load by adaptive batching. SM⁺_{CPU} horizontally scales the CPU variant. SM⁺_{GPU} horizontally scales a batch-1 optimized TensorRT variant (the cheapest GPU variant). We measured throughput every 2 seconds, and calculated the total cost. The cost for a running variant instance is estimated based on AWS EC2 pricing [17], proportional to its memory footprint. We normalize cost to 0.031 per GB/s for CPU, 0.190 per GB/s for Inferentia, and 0.498 per GB/s for GPU.

Workloads. We used three query patterns that are commonly observed in real-world setups [29]: (a) a flat, low load (4 QPS), (b) a steady, high load (slowly increase from 650 to 700 QPS), and (c) a fluctuating load (ranging between 4 and 80 QPS). Patterns (a) and (b) represent ideal cases for baselines, as they statically choose a variant; we used the most cost-effective CPU/GPU variant for each baseline.

Results and discussion. Figures 6a and 6d show the throughput and total cost, respectively, for INFaaS and the baselines when serving a flat, low load. While all systems met this low throughput demand, Clipper⁺_{GPU} and SM⁺_{GPU} incurred high costs since they use GPUs. INFaaS automatically selected CPU variants as they met the demand, thus reducing cost by $21.6\times$ and $21.3\times$ compared to Clipper⁺_{GPU} and SM⁺_{GPU}, respectively. For a steady, high load (Figures 6b and 6e), the observed throughput of Clipper⁺_{CPU} and SM⁺_{CPU} (about 10 QPS) was significantly lower than the demand. INFaaS automatically selected the batch-8 GPU variant, and both INFaaS and Clipper⁺_{GPU} met the throughput demand. While SM⁺_{GPU} replicated to 2 batch-1 GPU variants to meet the load, it was 5% more expensive than INFaaS and served 15% fewer QPS. Finally, for a fluctuating load (Figures 6c and 6f), INFaaS, Clipper⁺_{GPU}, and SM⁺_{GPU} met the throughput demand, while both SM⁺_{CPU} and Clipper⁺_{CPU} served only 10 QPS. During low load periods, INFaaS selected a CPU variant. At load spikes

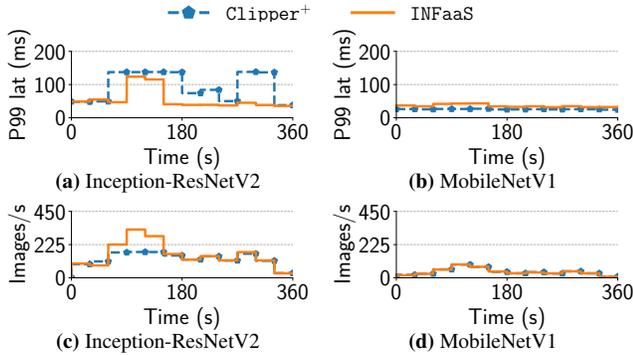


Figure 7: Performance of co-locating GPU model-variants when 80% of queries are served by Inception-ResNetV2.

(60-90 and 150-180 seconds), INFaaS upgraded to an Inferentia batch-1 variant. Hence, on average, INFaaS was $3\times$ cheaper than SM_{GPU}^+ and $Clipper^+_{GPU}$. If INFaaS were limited to CPU and GPU variants, it would still save $1.7\times$ cost over both the baselines. Similarly, even if we allowed baselines to use Inferentia, INFaaS would still save $1.9\times$ cost because baselines cannot dynamically switch between Inferentia and CPU variants. Figures 6a – 6c indicate that a single variant is neither the most cost-effective, nor the most performant for all scenarios. INFaaS achieves ease-of-use while (a) matching the baselines’ performance and cost in ideal cases (steady loads), and (b) outperforming the baselines during load changes. Thus, leveraging variants optimized for different hardware through model-vertical scaling, INFaaS is able to adapt to changes in load and query patterns, and improve cost by up to $21.6\times$ ($10\times$ on average).

6.3 Effectiveness of sharing resources

We now show how INFaaS manages and shares accelerators across models without affecting performance of queries. We found that co-locating models on an Inferentia chip did not cause noticeable interference, since variants can run on separate cores on the chip. Thus, we focus on evaluating GPU sharing. INFaaS detects when model-variants enter the *Overloaded/Interfered* state, and either migrates the model to a different GPU, or scales to a new GPU worker if all existing variants on the GPUs are in the *Overloaded/Interfered* state.

Experimental setup. We used the baseline of $Clipper^+$ with one model persisted on each GPU. Since $Clipper^+$ requires a pre-defined number of workers, we specified 2 GPU workers. For fairness, INFaaS started from one GPU and was allowed to scale up to 2 GPU workers. As noted in Section 2.3, the load at which sharing of GPUs starts affecting the performance negatively is different across models. We selected two model-variants that diverge in inference latency, throughput, and peak memory: Inception-ResNetV2 (large model) and MobileNetV1 (small model). Both variants are TensorRT-optimized for batch-1. We report throughput and P99 latency, measured every 30 seconds.

Workloads. To show the impact of model popularity on re-

source sharing, we evaluated a scenario with a popular model serving 80% QPS, and the other serving 20% QPS. We observed similar results with other popularity distributions or different models. We mapped the Twitter trace to a range between 50 and 500 QPS for a total of 75,000 batch-1 queries.

Results and discussion. Figure 7 shows P99 latency and throughput for both models when Inception-ResNetV2 is popular. When Inception-ResNetV2 and MobileNetV1 exceeded their profiled latencies, INFaaS marked them as interfered around 30 and 50 seconds, respectively. INFaaS started a new GPU worker (~ 30 seconds start-up latency), created an instance of each model on it, and spread the load for both models across the GPUs. The allocated resources for Inception-ResNetV2 with $Clipper^+$ were insufficient and led to a significant latency increase and throughput degradation. Unlike $Clipper^+$, INFaaS could further mitigate the latency increase by adding more GPU workers (limited to two in this experiment). Moreover, INFaaS saved 10% cost compared to $Clipper^+$ by (a) bin-packing requests across models to one GPU at low load, and (b) only adding GPUs when contentions were detected.

6.4 INFaaS’ decision overhead

On the critical path of serving a query, INFaaS makes the following decisions: (a) selecting a model-variant and (b) selecting a worker. Table 7 shows the median latency of making these decisions for INFaaS and the speedup over a brute-force search. Each row corresponds to a query specifying (1) a registered model and (2) application requirements. For each query, we show the decision latency when the selected variant was in (a) *Inactive*, *Overloaded*, or *Interfered* state, and (b) *Active* state. These decisions are made using the model-variant selection policy (Section 4.1).

When the registered model was explicitly specified, INFaaS incurred low overheads (~ 1 ms), as it needed to select only a worker. When the application requirements were provided, and the selected variant was not already loaded (State (a)), INFaaS selected a variant and the least-loaded worker for serving in 3.5ms. Otherwise, when the selected variant was already loaded (State (b)), INFaaS’ decision latency for selecting the variant and a worker was 2.2ms.

To measure scalability, we varied the number of model-variants from 10 to 166 (increments of 50); the latencies of Table 7 remain unchanged as the number of variants increases. This result was expected, since INFaaS’ Metadata Store uses constant access time data structures.

INFaaS keeps low overheads across its query submission modes: up to $44\times$ ($35.5\times$ on average) faster than brute-force.

7 Discussion

Failure Recovery. INFaaS’ VM-Autoscaler detects worker failures using RPC heartbeats, and starts a new worker with the state of the failed worker stored in the Metadata Store. For fault-tolerance, the controller is replicated using existing

Query	Variant Picked (Valid Options)	Median Latency in ms (Speedup vs brute-force)	
		State (a)	State (b)
resnet50-trt	resnet50-trt (1)	1.0 (N/A)	0.9 (N/A)
applD, >72%, 20ms	inceptionv3-trt (5)	3.5 (27×)	2.2 (44×)

Table 7: Median latency and speedup of making variant and worker selection decisions across 3 runs. State (a): variants are in the *Inactive* state, *Overloaded* state, or *Interfered* state. State (b): variants are in the *Active* state.

techniques [21, 37]. If the main controller fails, the incoming queries are re-routed to a standby controller. Since the Metadata Store is on the same machine as the controller, it is a part of the replicated state.

Query fairness. Using heterogeneous variants to serve queries means users may see different performance and accuracy results given the same query requirements, as INFaaS optimizes for cost-efficiency. However, INFaaS will always ensure the query requirements are met. INFaaS’ API is extensible to support further requirements for improved query fairness (e.g., bounding performance/accuracy variation [77]). **Explicitly controlling the runtime.** INFaaS’ model-less abstraction allows it to incorporate the ever-growing number of optimizers and runtimes. It also enables INFaaS’ model-selection algorithms to be extended separately. Explicitly controlling the runtime may allow INFaaS to provide more of a clear-box approach to optimizing inference serving, but may limit its generality and extensibility (e.g., supporting limited hardware platforms).

8 Related Work

Inference serving systems. TensorFlow Serving [5] provided one of the first production environments for models trained using the TensorFlow framework. Clipper [25] generalized it to enable the use of different frameworks and application-level SLOs. Pretzel [52], Nexus [66], and InferLine [24] built upon Clipper for optimizing inference serving pipelines. SageMaker [11], AI Platform [35], and Azure ML [1] offer developers inference services that autoscale VMs based on load. Triton Inference Server [6] optimizes GPU inference serving, supports CPU models, but requires static model instance configuration. DeepRecSys [39] statically optimizes batching and hardware selection for recommender systems, but requires developers to specify a variant, manage and scale model resources as the load varies. Clockwork [38] reduces GPU inference latency variability by ordering queries based on their SLOs and only running one query at a time. Model-Switching [77] switches between models with different accuracies during load spikes, while preserving the fraction of correct predictions returned within an SLO. It assumes pre-loaded models and does not consider heterogeneous hardware resources. Tolerance Tiers [42] allows developers to programmatically trade accuracy off for latency. None of these existing systems offer a simple model-less interface, like INFaaS, to navigate the variant search space on developers’ behalf, or dynamically leverage model-variants

to meet applications’ diverse requirements. However, prior work can be seen as complementary to INFaaS; e.g., INFaaS can adopt DeepRecSys’ recommender system optimizations and Clockwork’s predictable DNN worker.

Model-variant generators. Model graph optimizers [3, 12, 22, 72] perform optimizations, such as quantization and layer fusion, to improve latency and resource usage. However, developers still need to manually create and select variants, and manage the deployed variants. INFaaS uses these optimizers to create variants that can be used for meeting diverse application requirements, and automates model-variant selection for each query to minimize cost as load and resources vary.

Scaling. Autoscale [33] reviewed scaling techniques and argued for a simple approach that maintains the right amount of slack resources while meeting SLOs. Similarly, INFaaS’ autoscalers maintain headrooms and scale-down counters to cautiously scale resources. MARk [75] proposed SLO-aware model scheduling and scaling by using AWS Lambda to absorb unpredictable load bursts. Existing systems [1, 11, 35, 37] only support VM-level and model-horizontal scaling, while INFaaS introduces model-vertical scaling that leverages multiple diverse variants.

Sharing accelerators. NVIDIA MPS [57] enabled efficient sharing of GPUs that facilitated initial exploration into sharing GPUs for deep-learning. Existing systems [6, 27, 46, 74] also explored how to share GPUs spatially, temporally, or both. NVIDIA’s A100 GPUs support MIG [58]: hardware partitions and full isolation. AWS Inferentia supports spatial and temporal sharing via Neuron SDK [15]. INFaaS’ current implementation builds on Triton Inference Server (GPUs) and Neuron SDK (Inferentia), and provides SLO-aware accelerator sharing. INFaaS can also be extended to leverage other mechanisms for sharing additional hardware resources.

9 Conclusion

We presented INFaaS: an automated model-less system for distributed inference serving. INFaaS’ model-less interface allows application developers to specify high-level performance, cost or accuracy requirements for queries, leaving INFaaS to select and deploy the model-variant, hardware, and scaling configuration. INFaaS automatically provisions and manages resources for serving inference queries to meet their high-level goals. We demonstrated that INFaaS’ model-variant selection policy and resource sharing leads to reduced costs, better throughput, and fewer SLO violations compared to state-of-the-art inference serving systems.

Acknowledgments

We thank our shepherd, Sangeetha Abdu Jyothi, and the anonymous reviewers for their helpful feedback. We thank Honglin Yuan, Hilal Asi, Peter Kraft, Matei Zaharia, John Wilkes, and members of the MAST research group for their insightful discussions to improve this work. This work was supported by the Stanford Platform Lab and its industrial affiliates, the SRC Jump program (CRISP center), and Huawei.

References

- [1] *Azure Machine Learning*, 2018. <https://docs.microsoft.com/en-us/azure/machine-learning/>.
- [2] *gRPC*, 2018. <https://grpc.io/>.
- [3] *NVIDIA TensorRT: Programmable Inference Accelerator*, 2018. <https://developer.nvidia.com/tensorrt>.
- [4] *Redox*, 2018. <https://github.com/hmartiro/redox>.
- [5] *TensorFlow Serving for model deployment in production*, 2018. <https://www.tensorflow.org/serving/>.
- [6] *NVIDIA Triton Inference Server*, 2020. <https://github.com/triton-inference-server/server>.
- [7] Hanguang-800 NPU. <https://www.t-head.cn/product/npu>.
- [8] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, March 2017. USENIX Association.
- [9] Amazon EC2. <https://aws.amazon.com/ec2/>, 2018.
- [10] Amazon S3. <https://aws.amazon.com/s3/>, 2018.
- [11] Amazon SageMaker. <https://aws.amazon.com/sagemaker/>, 2018.
- [12] Amazon SageMaker Neo. <https://aws.amazon.com/sagemaker/neo/>, 2018.
- [13] Twitter Streaming Traces. <https://archive.org/details/archiveteam-twitter-stream-2018-04>, 2018.
- [14] Mohammed Attia, Younes Samih, Ali Elkahky, and Laura Kallmeyer. Multilingual multi-class sentiment classification using convolutional neural networks. pages 635–640, Miyazaki, Japan, 2018.
- [15] AWS Neuron. <https://github.com/aws/aws-neuron-sdk>.
- [16] Deliver high performance ML inference with AWS Inferentia. https://dl.awsstatic.com/events/reinvent/2019/REPEAT_1_Deliver_high_performance_ML_inference_with_AWS_Inferentia_CMP324-R1.pdf.
- [17] AWS EC2 Pricing. <https://aws.amazon.com/ec2/pricing/on-demand/>, 2018.
- [18] AWS Inferentia. <https://aws.amazon.com/machine-learning/inferentia/>, 2018.
- [19] Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napolitano. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6:64270–64277, 2018.
- [20] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):10, 2016.
- [21] Prima Chairunnanda, Khuzaima Daudjee, and M. Tamer Özsu. Confluxdb: Multi-master replication for partitioned snapshot isolation databases. *PVLDB*, 7:947–958, 2014.
- [22] 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.
- [23] Clipper. <https://github.com/ucbrise/clipper>.
- [24] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 477–491, 2020.
- [25] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27–29, 2017*, pages 613–627, 2017.
- [26] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. Hydra: a federated resource manager for data-center scale analytics. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 177–192, Boston, MA, February 2019. USENIX Association.
- [27] Abdul Dakkak, Cheng Li, Simon Garcia De Gonzalo, Jinjun Xiong, and Wen-Mei W. Hwu. Trims: Transparent and isolated model sharing for low latency deep learning inference in function as a service environments. *CoRR*, abs/1811.09732, 2018.
- [28] Thomas Davidson, Dana Warmusley, Michael Macy, and Ingmar Weber. Automated Hate Speech Detection and the Problem of Offensive Language. In *Proceedings of the Eleventh International AAAI Conference on Web and Social Media (ICWSM 2017)*, 2017.
- [29] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14*, pages 127–144, New York, NY, USA, 2014. ACM.
- [30] Jia Deng, Wei Dong, Richard Socher, Li jia Li, Kai Li, and Li Fei-fei. Imagenet: A large-scale hierarchical image database. In *In CVPR*, 2009.
- [31] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12*, pages 99–112, New York, NY, USA, 2012. ACM.
- [32] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Masesgill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture, ISCA '18*, pages 1–14, Piscataway, NJ, USA, 2018. IEEE Press.
- [33] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems (TOCS)*, 30(4):14, 2012.
- [34] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.
- [35] Google Cloud AI Platform. <https://cloud.google.com/ai-platform/>, 2018.
- [36] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeong-jae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, Boston, MA, 2019. USENIX Association.
- [37] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S McKinley, and Björn B Brandenburg. Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency. In *Proceedings of the 18th ACM/IIFIP/USENIX Middleware Conference*, pages 109–120. ACM, 2017.
- [38] 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. USENIX Association, November 2020.

- [39] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. DeepRecSys: A System for Optimizing End-To-End At-scale Neural Recommendation Inference, 2020.
- [40] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The Architectural Implications of Facebook’s DNN-Based Personalized Recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–501, Feb 2020.
- [41] LLC Gurobi Optimization. Gurobi Optimizer Reference Manual, 2020.
- [42] M. Halpern, B. Boroujerdian, T. Mummert, E. Duesterwald, and V. Reddi. One size does not fit all: Quantifying and exposing the accuracy-latency trade-off in machine learning cloud service apis via tolerance tiers. In *Proceedings of the 19th International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019.
- [43] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, HPCA ’18. IEEE, 2018.
- [44] 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.
- [45] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 269–286, Carlsbad, CA, October 2018. USENIX Association.
- [46] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. Dynamic space-time scheduling for gpu inference. In *LearningSys Workshop at Neural Information Processing Systems 2018*, 2018.
- [47] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: Scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM ’18, pages 253–266, New York, NY, USA, 2018. ACM.
- [48] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC ’17, pages 445–451, New York, NY, USA, 2017. ACM.
- [49] 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, pages 1–12, New York, NY, USA, 2017. ACM.
- [50] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: Optimizing neural network queries over video at scale. *Proc. VLDB Endow.*, 10(11):1586–1597, August 2017.
- [51] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, 2018. USENIX Association.
- [52] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626, Carlsbad, CA, 2018. USENIX Association.
- [53] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP ’75, page 132–140, New York, NY, USA, 1975. Association for Computing Machinery.
- [54] Xin Li, Zhuzhong Qian, Sanglu Lu, and Jie Wu. Energy efficient virtual machine placement algorithm with balanced and improved resource utilization in a data center. *Mathematical and Computer Modelling*, 58(5-6):1222–1235, 2013.
- [55] 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, Santa Clara, CA, February 2020. USENIX Association.
- [56] Fatemehsadat Mireshghallah, Mohammadkazem Taram, Prakash Ramrakhiani, Ali Jalali, Dean Tullsen, and Hadi Esmaeilzadeh. Shredder: Learning noise distributions to protect inference privacy. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’20, page 3–18, New York, NY, USA, 2020. Association for Computing Machinery.
- [57] NVIDIA MPS. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2018.
- [58] NVIDIA Multi-instance GPU. <https://www.nvidia.com/en-us/technologies/multi-instance-gpu/>, 2020.
- [59] Young H. Oh, Quan Quan, Daeyeon Kim, Seonghak Kim, Jun Heo, Sungjun Jung, Jaeyoung Jang, and Jae W. Lee. A portable, automatic data quantizer for deep neural networks. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT ’18, pages 17:1–17:14, New York, NY, USA, 2018. ACM.
- [60] Alex Poms, Will Crichton, Pat Hanrahan, and Kayvon Fatahalian. Scanner: Efficient video analysis at scale. *ACM Transactions on Graphics (TOG)*, 37(4):1–13, 2018.
- [61] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459, 2020.

- [62] Redis. <https://redis.io>, 2018.
- [63] Francisco Romero and Christina Delimitrou. Mage: Online and interference-aware scheduling for multi-scale heterogeneous systems. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT '18*, New York, NY, USA, 2018. Association for Computing Machinery.
- [64] Steven S. Seiden. On the online bin packing problem. *J. ACM*, 49(5):640–671, September 2002.
- [65] Mohammad Shahradd, Rodrigo Fonseca, Inigo Goiri, Gohar Irfan, 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)*, Boston, MA, USA, July 2020. USENIX Association. To Appear.
- [66] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19*, page 322–337, New York, NY, USA, 2019. Association for Computing Machinery.
- [67] Leonid Velikovich, Ian Williams, Justin Scheiner, Petar S. Aleksic, Pedro J. Moreno, and Michael Riley. Semantic lattice processing in contextual automatic speech recognition for google assistant. In *Interspeech 2018, 19th Annual Conference of the International Speech Communication Association, Hyderabad, India, 2-6 September 2018.*, pages 2222–2226, 2018.
- [68] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, Santa Clara, CA, 2016. USENIX Association.
- [69] Joachim von zur Gathen and Malte Sieveking. A bound on solutions of linear integer equalities and inequalities. *Proceedings of the American Mathematical Society*, 72(1):155–158, 1978.
- [70] ACL 2016 First Conference on Machine Translation (WMT16). <http://www.statmt.org/wmt16/>.
- [71] 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, 2018. USENIX Association.
- [72] XLA: Optimizing Compiler for Machine Learning. <https://www.tensorflow.org/xla>.
- [73] Neeraja J. Yadwadkar, Francisco Romero, Qian Li, and Christos Kozyrakis. A case for managed and model-less inference serving. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS '19*, page 184–191, New York, NY, USA, 2019. Association for Computing Machinery.
- [74] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained GPU sharing primitives for deep learning applications. *CoRR*, abs/1902.04610, 2019.
- [75] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, Renton, WA, July 2019. USENIX Association.
- [76] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 377–392, Boston, MA, March 2017. USENIX Association.
- [77] Jeff Zhang, Sameh Elnikety, Shuayb Zarar, Atul Gupta, and Siddharth Garg. Model-switching: Dealing with fluctuating workloads in machine-learning-as-a-service systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, July 2020.

Jump-Starting Multivariate Time Series Anomaly Detection for Online Service Systems

Minghua Ma
Tsinghua University, BNRist

Shenglin Zhang^{*}
Nankai University

Junjie Chen
Tianjin University

Jun Xu
Georgia Tech

Haozhe Li, Yongliang Lin
Nankai University

Xiaohui Nie
Tsinghua University, BNRist

Bo Zhou, Yong Wang
CNCERT/CC

Dan Pei
Tsinghua University, BNRist

Abstract

With the booming of online service systems, anomaly detection on multivariate time series, such as a combination of CPU utilization, average response time, and requests per second, is important for system reliability. Although a collection of learning-based approaches have been designed for this purpose, our empirical study shows that these approaches suffer from long *initialization time* for sufficient training data. In this paper, we introduce the *Compressed Sensing* technique to multivariate time series anomaly detection for rapid initialization. To build a jump-starting anomaly detector, we propose an approach named *JumpStarter*. Based on domain-specific insights, we design a shape-based clustering algorithm as well as an outlier-resistant sampling algorithm for *JumpStarter*. With real-world multivariate time series datasets collected from two Internet companies, our results show that *JumpStarter* achieves an average F1 score of 94.12%, significantly outperforming the state-of-the-art anomaly detection algorithms, with a much shorter initialization time of twenty minutes. We have applied *JumpStarter* in online service systems and gained useful lessons in real-world scenarios.

1 Introduction

In recent years, online service systems based on cloud computing, *e.g.*, online office, e-commerce, are becoming increasingly popular. For example, the number of daily meeting participants of the video conferencing app Zoom jumps to over 300 million before May 2020 [31]. Due to the complexity and the large scale of online service systems, automatic anomaly detection is of ultimate importance to guarantee their reliability [36]. To closely monitor the quality of service, online service providers or cloud computing platforms, such as Microsoft and AWS, continuously collect the monitoring data of each performance metric (*e.g.*, CPU utilization, average response time, and requests per second) at equally spaced

intervals [3, 33]. The monitoring data of a metric form a univariate time series, and thus that of a service system, which has multiple metrics, constitutes a multivariate time series.

Traditional multivariate time series anomaly detection approaches are typically based on detecting univariate time series [15, 32, 36]. However, operators are concerned about the status of the overall service rather than that of a specific metric [28]. Because the univariate time series anomaly detection cannot capture the complex temporal relationships among different univariate time series [28], they tend to cause alert storms [5]. To address this problem, recent works [12, 22–24, 28, 34] use deep learning techniques to build learning models for multivariate time series anomaly detection. For example, the state-of-the-art approach, OmniAnomaly [28], utilizes a stochastic recurrent neural network model to learn the temporal relationships of multivariate time series.

Learning-based approaches are hardly applicable in practice because they usually require a long period of training data. Online service systems are deployed or changed very frequently to deploy new features, fix bugs [35], *etc.* In large service providers, such as Google [3] and Baidu [35], it is reported that thousands of software changes are deployed every day. Due to these software changes, the data distribution of multivariate time series can change dramatically, which is called the expected concept drift [17]. For example, when operators conduct a software change to deploy a service to more instances, the metric “Requests Per Second” in each instance will drop significantly as shown in Figure 1. This is expected to operators, and they do not need to roll back the software change. After the change, this will cause a lot of false alarms or false positives, since it invalidates the learning-based anomaly detection models trained based on the data before the change [17]. It is because the common assumption in deep learning that the data distribution must remain the same across training and test set [10] is violated. Therefore, these learning-based approaches have to be retrained. This retraining process, however, can consume tens to hundreds of days [28, 32] before reaching steady state.

To quantitatively measure how long it takes to “initialize”

^{*}Shenglin Zhang is the corresponding author.

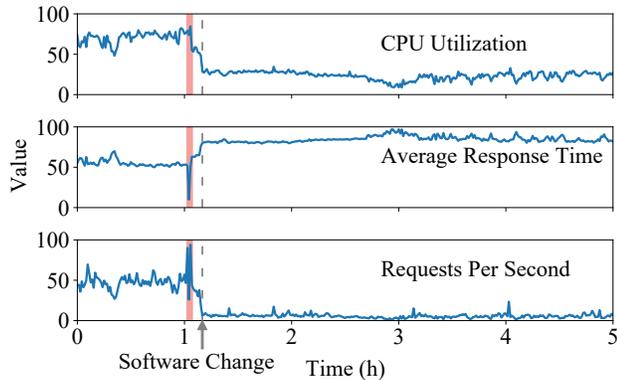


Figure 1: The multivariate time series (selected as examples) of an online service system before and after a software change. The red segment is labeled anomalous.

an anomaly detection model for multivariate time series, we first define *initialization time*, as the time lag between when the model is launched and when it becomes well trained. We then conduct an empirical study based on the datasets collected from real-world online service systems. Through this study, we summarize two key findings: (1) The average initialization time of existing learning-based approaches ranges from ten days to one hundred days. (2) The state-of-the-art approaches (*i.e.*, OmniAnomaly [28] and MSCRED [34]) do not achieve a satisfactory performance when they are improved with *incremental retraining* [15]. When the period of training data is short, the accuracy of these approaches are low. Therefore, we aim to design a robust anomaly detection approach for multivariate time series with small initialization time.

In this paper, we propose a novel multivariate time series anomaly detection approach, called *JumpStarter*, that is based on *Compressed Sensing* (CS). CS is a signal processing technique where high-energy components in a matrix (multivariate time series) are sparse (*i.e.*, have few high-energy components) [4]. Hence, the difference between the original and the reconstructed multivariate time series, comprised only of low-energy components, should resemble white noise, when the original time series contains no anomaly. The intuition behind using CS for anomaly detection is that anomalies in multivariate time series, such as jitters, sudden drops or surges, usually manifest themselves as strong signals that contain high-energy components, which would differ significantly from white noise [16]. Hence we can tell whether a time series contains anomalies by checking whether the difference between the original and the reconstructed multivariate time series in a sliding window [32] looks very differently from white noise. Since CS only uses a fixed-length window to “train” an anomaly detection model, the initialization time of *JumpStarter* depends on the window size, which is typically twenty minutes. It is much shorter than the initialization time of learning-based approaches. We now provide an intuitive explanation of why CS-based *JumpStarter* requires

much less training data than those learning-based approaches. A learning-based approach has to *explicitly* learn the “behavior” (probability distribution) of a normal multivariate time series in order to detect anomalies. In comparison, in *JumpStarter* the reconstructed multivariate time series of *implicitly* inherits this normal behavior without involving any explicit learning. Due to the complexity of real-world scenarios, however, it is challenging to apply CS to implement a jump-starting multivariate time series anomaly detection in the following situations:

Large number of time series. In large-scale online service systems, tens of time series are monitored for each system forming a multivariate time series [17]. Typically, it is time-consuming for CS to reconstruct a large number of time series, because it needs to solve a convex optimization problem whose time complexity depends on the number of time series. To tackle this challenge, we cluster these time series based on shape. Since the time series in the same group exhibit a similar shape, they can be reconstructed efficiently without losing temporal relationships [14].

Sampling from random anomalous segments in time series. CS typically uses a Gaussian distribution to sample from the original time series to guarantee Restricted Isometry Property (RIP). Nevertheless, it may inevitably generate reconstructed time series sampled from some long-lasting anomalous segments. The anomaly detection model based on these reconstructed time series can cause some false positives and/or false negatives. Therefore, we design an outlier-resistant sampling algorithm to sample from normal time series segments rather than anomalous ones.

We conducted a comprehensive study to evaluate the performance of *JumpStarter* based on three datasets from 28 and 30 large-scale industrial online service systems of two Internet companies, respectively. The first dataset, from company *A*, is a 5-week-long time-series open dataset. The other two datasets are from a top-tier global content platform company *B*, which provides service for more than 800 million users around the world. These datasets contain 7-week-long time series. Our experimental results illustrate that *JumpStarter* outperforms both the state-of-the-art learning-based multivariate time series anomaly detection approaches (*i.e.*, OmniAnomaly [28] and MSCRED [34]) clustering-based LESINN [20]. Also, *JumpStarter* is more suitable than RRCF [11] used in AWS CloudWatch. The average F1 score of *JumpStarter* is 94.12%, while those of the other four approaches are 86.51%, 59.64%, 82.5%, and 36.01% respectively. Also, our results demonstrate that the main components in *JumpStarter* (*i.e.*, shape-based clustering and outlier-resistant sampling) significantly contribute to the overall performance of *JumpStarter*. *JumpStarter* achieves good accuracy with an initialization time as short as twenty minutes, open sourced at <https://github.com/NetManAIops/JumpStarter>. We applied *JumpStarter* to *B* and it indeed achieved a good anomaly detection performance in practice. We also present two cases

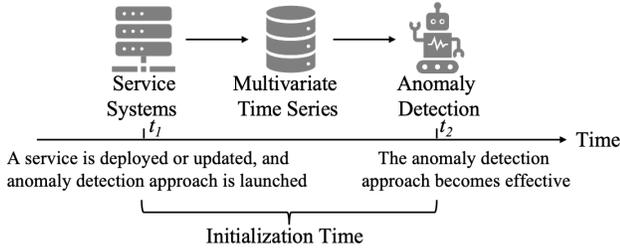


Figure 2: The initialization time of an anomaly detector

and gain lessons for both academia and industry.

2 Background and Empirical Study

2.1 Background

Multivariate time series. In an online service system, operators continuously collect monitoring data of multiple metrics or extract numerical values from logs [37]. A service level metric (*e.g.*, average response time), or a machine level metric (*e.g.*, CPU utilization, memory utilization), is usually collected by equal interval, forming a univariate time series. Any univariate time series alone, however, cannot capture all types of system’s performance issues [28]. Because a system typically has a collection of monitoring metrics, it can be denoted as a multivariate time series [17], which includes diverse types of univariate time series and thus track various aspects of performance issues. With the scale and complexity of the system increasing, it is becoming more difficult to manually inspect system anomalies. Therefore, multivariate time series anomaly detection is of great importance [28, 34]. We denote a multivariate time series at time t as $\mathbf{X}_t = [x_t^1, x_t^2, \dots, x_t^n]^T$, where $x_t^i = [x_{t-w+1}^i, x_{t-w+2}^i, \dots, x_t^i]$ is the univariate time series of the i^{th} monitoring metric, n is the number of metrics, and w is the observation window size. We apply the sliding window, which is a common practice in time series anomaly detection [32, 35], to construct \mathbf{X}_t .

Anomaly detection. Anomaly detection using multivariate time series [28] is important in online service systems. In previous anomaly detection works [12, 22–24, 28, 34], operators have a rough consensus on the following points: 1) A multivariate time series anomaly is a data point or a data segment that significantly deviates from operators’ expectations of normal behavior, and it can be visually observed (*e.g.*, in Figure 1). 2) An anomaly indicates something might have gone wrong, although further investigation may still be needed for verification. 3) Anomaly detection is often used as a failure discovery mechanism. Formally, we define multivariate time series anomaly detection: for time t , given its multivariate time series \mathbf{X}_t , we determine whether an anomaly occurs (*e.g.*, jitter, sudden drop or surge), which is denoted by $y_t = 1$ if yes and $y_t = 0$ otherwise.

Table 1: Comparison of the initialization time (days) on three datasets (S1~S3) used in their works. * denotes univariate time series anomaly detector, which can be used for multivariate time series by combining it with majority vote [28].

Approach	S1	S2	S3	Avg.
MSCRED [34]	7	13	-	10
OmniAnomaly [28]	17	15	17	16.3
LSTM-NDT [12]	69	36	-	52.5
* Opprentice [15]	56	56	56	56
* Donut [32]	102	110	99	103.6

2.2 An Empirical Study on Initialization Time

Anomaly detection initialization time. With a new service being deployed or updated, operators usually launch an anomaly detection approach for it. The initialization time of the anomaly detection approach is the time lag between when it is launched (t_1) and when it becomes effective (t_2), as shown in Figure 2. Many prior approaches, *e.g.*, [12, 22, 23, 28, 34], use a learning-based workflow to detect anomalies. Typically, they are periodically trained based on historical data [15]. The initialization time of these approaches, *e.g.*, tens of days, is relatively long, because they usually need to offer a lot of historical data for training. In Table 1 we list the suggested initialization time of five learning-based anomaly detection approaches on different datasets. For example, OmniAnomaly [28] used two robot system datasets (denoted as S1, S2) and a server dataset (S3, which also used in our experiment as D1). From the last column of Table 1, we can see that the average initialization time of these approaches ranges from 10 days to more than one hundred days, indicating that it is unsuitable to use these approaches for newly deployed or updated systems.

Incremental retraining. Considering the long initialization time of learning-based anomaly detection approaches, one may suggest incremental retaining, *i.e.*, gradually (incrementally) adding a short-period (say one day) of data to train these approaches. In this way, we can improve the performance of these approaches step by step. Adding one day’s data each time is because these learning-based approaches need at least thousands of data points to converge [32]. We then try to apply incremental retraining to the state-of-the-art multivariate time series anomaly detection approaches, *i.e.*, OmniAnomaly [28] and MECRED [34]. The dataset is the same as what is used in OmniAnomaly (see §4.1 for more details). We gradually enlarge the training set from one day’s data to 13 days’ data (*i.e.*, the largest training set of this dataset), and the testing set remains as the data collected after the 13th day.

This sounds ideal, but anomaly detection using incremental retraining cannot ensure satisfactory performance. Figure 3 shows the average F1 score and training time of OmniAnomaly and MECRED as the period of training data increases (day by day), respectively. From Figure 3(a), we can see that the average F1 scores of both OmniAnomaly and MECRED increase along with more training data being used, and

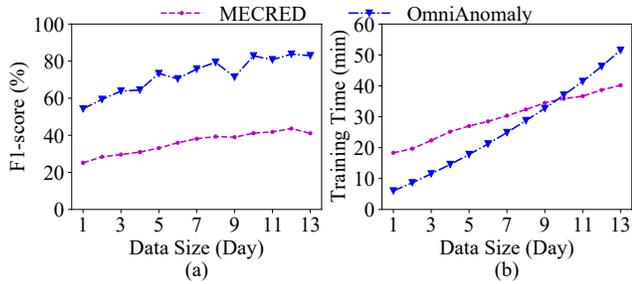


Figure 3: Performance of OmniAnomaly [28] and MECRED [34] by incremental retraining.

they do not converge until 10 days’ data is used for training. One primary reason is that these learning-based approaches have to *explicitly* learn the probability distribution of a multivariate time series from a large amount of training data to capture its normal behavior. Figure 3(b) shows that the training time of both OmniAnomaly and MECRED increases linearly with the size of training data. When the training dataset contains 10 days of data, it takes about 35 minutes to train OmniAnomaly or MECRED. Therefore, these approaches are not suitable for newly deployed or updated systems due to their non-robustness and considerable training cost.

3 JumpStarter Approach

3.1 Key Idea and Challenges

To deal with the aforementioned limitations of learning-based approaches, we propose to use Compressed Sensing (CS) [9] for multivariate time series anomaly detection. CS is a signal processing technique for reconstructing a signal from a series of sampling measurements [8]. The signal reconstructed from these samples preserves the high-energy components of the original signal with high probability under some mild assumptions [8]. As explained earlier, we can detect anomalies in a multivariate time series by checking whether the reconstructed signal differs from the original signal (multivariate time series) by more than white noise. Since CS does not require any training, the initialization time of the CS-based anomaly detection for \mathbf{X}_t is the window size w .

Two Strawman Solutions using CS. Intuitively, we can apply CS to reconstruct \mathbf{X}_t in two ways: treating \mathbf{X}_t as a whole $n \times w$ matrix, or as n separate univariate time series. In the former way, we randomly sample some data from \mathbf{X}_t and then reconstruct it following [26] (more details to be provided in §3.5). Figure 4(a) shows the reconstructed multivariate time series. As shaded in Figure 4(a), there is a significant difference between the original (blue lines) and the reconstructed multivariate time series (dashed brown lines) when an anomaly occurs. This is a desired behavior (of CS). However, the first two reconstructed time series fluctuate frequently all the time, whereas both original time series are stable except

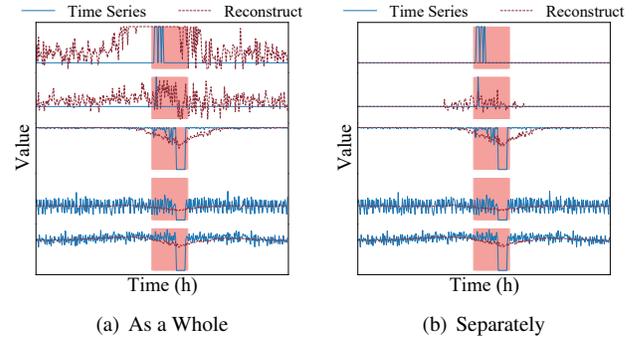


Figure 4: Examples of CS-based anomaly detection when the multivariate time series is reconstructed as a whole $n \times w$ matrix (a) or as n separate univariate time series (b). The red-shaded regions denote the anomalies labeled by operators.

during the anomaly window (shaded). This is an undesired behavior that indicates inaccurate reconstruction. In the latter way, as shown in Figure 4(b), the difference between the original and reconstructed univariate time series manifests as white noise in normal segments and as large fluctuations in anomalous ones, which accurately captures the anomalies for each *univariate time series*. However, it cannot capture the complex relationships among multivariate time series [28]. Moreover, due to the challenge of a large number of univariate time series, the separate reconstruction is more computationally expensive.

Problem of Random Gaussian Sampling. The first step of our CS approach is to sample from a multivariate time series. The sampled matrix needs to guarantee Restricted Isometry Property (RIP) [4] so that it can reconstruct the original multivariate time series properly. It has been shown that random Gaussian sampling satisfies RIP [9]. Random sampling, however, samples some data points also from anomalous segments. In this case, the reconstructed multivariate time series will not be significantly different from the original one when a system becomes anomalous. Thus it inevitably degrades the anomaly detection performance.

3.2 Overview of JumpStarter

The *JumpStarter* approach, which consists of both offline and online processing procedures, is shown in Figure 5. To tackle the challenge of a large number of time series, we adopt a *shape-based clustering* method to group the univariate time series of a multivariate time series into several groups in the offline processing. The sliding window technique is applied to multivariate time series in the online anomaly detection. For each group of univariate time series, we propose a novel *outlier-resistant sampling* algorithm to solve the challenge introduced by sampling from anomalous segments, and apply *compressed sensing* to reconstruct them. After that, we concatenate these reconstructed time series, measure the differ-

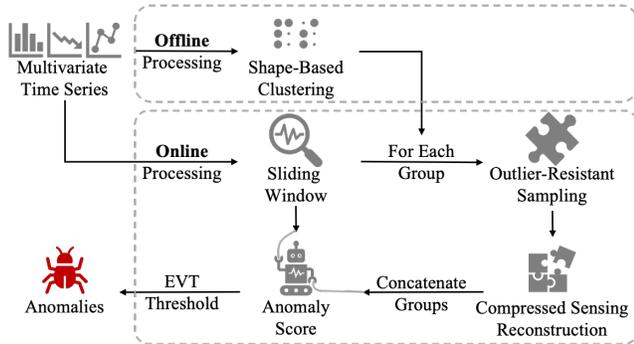


Figure 5: *JumpStarter* approach consists of offline processing and online processing, of which output is whether anomaly or not.

ence between the original and the reconstructed multivariate time series as an anomaly score, and detect anomalies using *EVT threshold* against the anomaly score.

3.3 Shape-Based Clustering

While each strawman solutions has its advantages and drawbacks, we find a way to combine them to get the better of both. Specifically, we split \mathbf{X}_t into several clusters of time series and reconstruct each cluster. The question is, how this splitting should be done. Recall the first strawman solution cannot deal with time series of different shapes. Hence, we choose to split \mathbf{X}_t based on their shape. This solution can achieve both high accuracy and high efficiency.

We adopt shape-based distance [21], a cross correlation-based method, to measure the distance between two univariate time series. It can achieve high computational efficiency when dealing with high-dimensional time series. Different from ROCKA [14], we use hierarchical clustering, which is efficient and need not manually configure the number of clusters, as the base clustering algorithm. Table 2 illustrates an example of clustering results. The nine univariate time series of a multivariate time series are grouped into three clusters. In each cluster, the time series are correlated to the physical meaning of their corresponding monitoring metrics, demonstrating that our method is intuitive.

We cluster univariate time series for every multivariate time series based on one-day worth of data, because most univariate time series are roughly periodical with a 24-hour cycle that coincides with customers’ diurnal usage pattern [17]. Moreover, we observe that the shape of a univariate time series usually remains unchanged after a software change [38]. As a result, it has no need to re-cluster after a software change.

3.4 Outlier-Resistant Sampling

Insight. To solve the problem of random Gaussian sampling, we gain insight from the investigation of a large number

Table 2: An example of clustering the multivariate time series into three clusters. Each univariate time series is named based on its corresponding monitoring metric.

#	Cluster of Univariate Time Series	Explanation
1	rx-pkts-eth0, rx-bytes-eth0	# received packets/bytes
2	tcp-insegs, tcp-outsegs, tx-pkts-eth0	TCP network metrics
3	cpu-ctxt, cpu-user, cpu-system, cpu-nice	CPU utilization metrics

of online service systems and the discussion with operators. Anomalies rarely occur in real-world scenarios [15, 32]. That is, anomalies are usually outliers in an observation (sliding) window [16]. If an anomaly lasts longer than the window size, it can be captured from the beginning since it is significantly different from the normal pattern. Therefore, we can adopt a simple outlier detection algorithm to obtain the *sampling confidence* of each data point. The higher a data point is likely to be an outlier, the lower its sampling confidence is, and the less likely it will be selected. Based on this insight, we design an outlier-resistant sampling algorithm, *i.e.*, one-dimensional random Gaussian, which not only guarantees RIP but also resists outliers.

Algorithm. We describe the design of outlier-resistant sampling in Algorithm 1 and show the main steps in Figure 6. After the shape-based clustering, for each cluster, we can obtain an $w * k$ matrix, which is constituted of k univariate time series. We also configure a sampling ratio, θ , as the input of the algorithm. To begin with, from Line 1 to 4 of Algorithm 1, we initialize a zero $m * w$ matrix \mathbf{T} , where $m = \lceil w * \theta \rceil$. ϕ is initialized as a vector containing m random Gaussian sample timestamps ranging from 0 to w . Motivated by [20], we adopt a light-weight algorithm LESINN to calculate the sampling confidence vector \mathbf{sc} , which determines the sampling confidence of each timestamp. Nevertheless, as demonstrated in §4.2, LESINN itself cannot effectively handle multivariate time series anomaly detection because it cannot capture the complex temporal relationships among these time series. Figure 6(a) shows one example of the original time series (blue line) with a window size of 20 and its sampling confidence (orange dotted line) for each timestamp.

Then, we perform value sampling (Line 5 to 14). We first normalize \mathbf{sc} to make its values sum up to one, and create R equal-width *steps*. Each *step* is mapped to a timestamp t based on the normalized \mathbf{sc} (Line 8). For each *step*, we randomly sample a value from $(0, 1)$, and compare it with the probability of a similar version of Gaussian distribution:

$$P_i(step) = \rho \cdot \exp\left(-\frac{(\phi_i - step)^2}{2\sigma^2}\right) \quad (1)$$

Note that ρ is a parameter representing the max sampling points (height in Figure 6(b)). For more theoretical details, please refer to the work of Barranca, *et al.* [2]. σ is the standard deviation of the distribution. If the random value is smaller than $P(step)$, we add one to $\mathbf{T}[i][t]$. Note that after

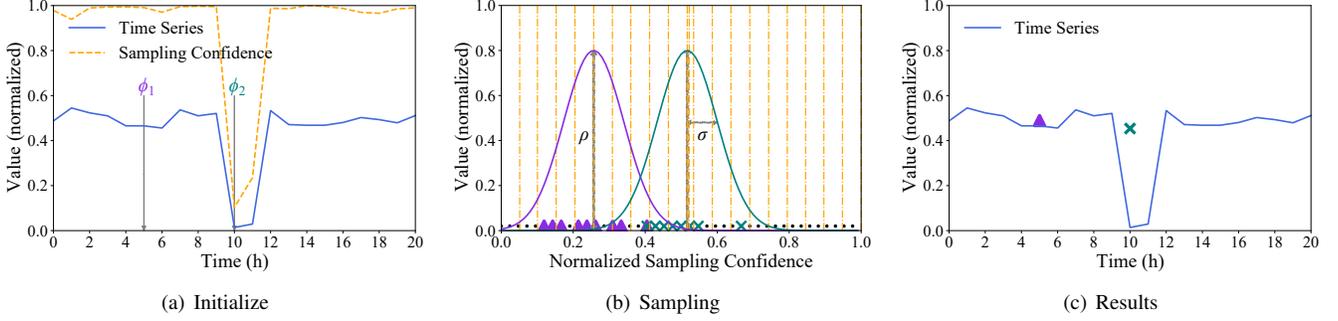


Figure 6: A toy example of outlier-resistant sampling. In (a), to make it simple, we just plot one time series whose values are normalized, and it has an anomalous segment ranging from (timestamp) 9 to 12. We set $m = 2$ in this example. In (b), the small black dot is steps, and the purple and green marks are “selected” steps. (c) shows two sampled data points.

all the iterations, the randomly sampled timestamps in ϕ may not be “shot” by t . Therefore, we add one to $\mathbf{T}[i][\phi_i]$ for each sampled timestamp i . Figure 6(b) shows a concrete example of sampling values. In this figure, there are $R = 42$ steps (the small points in the bottom), and the normalized \mathbf{sc} (the orange box) is shown in its original temporal order, which represents a timestamp.

Finally, we normalize each column of \mathbf{T} (Line 15 to 18). The sampling matrix \mathbf{B} is the dot product of \mathbf{T} and \mathbf{X}_t^c . As shown in Figure 6(c), the purple dot and green cross are the sampled data points in \mathbf{B} . We can see that our algorithm is resistant to anomalous data points because the sampled green cross point represents “normal pattern” even though its corresponding point in the original time series is anomalous. Note that although we apply LESINN to calculate the sampling confidence vector, which is of great importance to the resistant outlier, our algorithm is also robust to other sampling confidence measurements beyond LESINN.

3.5 Compressed Sensing Reconstruction

Here we provide a quick introduction to the signal reconstruction step in compressed sensing, using this anomaly detection problem as the context. The objective of compressed sensing reconstruction is to “solve” [4]:

$$\mathbf{A}\mathbf{X}'_t = \mathbf{B} \quad (2)$$

where \mathbf{X}'_t is the multivariate time series to be reconstructed from \mathbf{B} that is sampled from the original \mathbf{X}_t . We put quotations around the word “solve”, since this equation is not solvable in the usual sense, as it is under-determined. Rather, ideally we would like to compute the sparsest (*i.e.*, containing the smallest number of nonzero components) such \mathbf{X}'_t . However, this computation, known as L^0 minimization, is NP-complete. A classic CS result is that, when \mathbf{X}_t is sparse, minimizing the L^1 of \mathbf{X}'_t (while satisfying Equation 2) results in the same solution as the L^0 minimization with high probability [4].

Algorithm 1: Outlier-Resistant Sampling

Input: $\mathbf{X}_t^c(w * k)$: k univariate time series of \mathbf{X}_t in cluster c ,
 θ : Initial sampling ratio
Output: Sampling matrix $\mathbf{B}(m * k)$

- 1 $m \leftarrow \lceil w * \theta \rceil$
- 2 $\mathbf{T}(m * w) \leftarrow 0$
- 3 $\phi(m * 1) \leftarrow$ randomly sampling m timestamps from $[0, w]$
- 4 $\mathbf{sc}(w * 1) \leftarrow \mathbf{SamplingConfidence}(\mathbf{X}_w)$
- 5 $\mathbf{sc} \leftarrow \frac{\mathbf{sc}}{\sum_{j=0}^{w-1} \mathbf{sc}_j}$
- 6 $step = 0$
- 7 **while** $step \leq 1$ **do**
- 8 $t \leftarrow \arg \min_{0 \leq a < w} step < \sum_{j=0}^a \mathbf{sc}(j)$
- 9 **foreach** $i \in [0, m)$ **do**
- 10 **if** $\mathbf{random}(0, 1) < P_t(step)$ **then**
- 11 $\mathbf{T}[i][t] \leftarrow \mathbf{T}[i][t] + 1$
- 12 $step \leftarrow step + 1/R$
- 13 **foreach** $i \in [0, m)$ **do**
- 14 $\mathbf{T}[i][\phi_i] \leftarrow \mathbf{T}[i][\phi_i] + 1$
- 15 **foreach** $i \in [0, m)$ **do**
- 16 $\mathbf{T}[i] \leftarrow \mathbf{T}[i] / \sum_{j=0}^{w-1} \mathbf{T}[i][j]$
- 17 $\mathbf{B} \leftarrow \mathbf{T} \cdot \mathbf{X}_t^c$
- 18 **return** \mathbf{B}

The sampling matrix \mathbf{A} is calculated as:

$$\mathbf{A} = \phi(\mathbf{D} \otimes \mathbf{D}^T) \quad (3)$$

where \mathbf{D} is the inverse discrete cosine transform [13] of the original time series \mathbf{X}_t^c , and \otimes is the Kronecker product [29].

To solve Equation 2 (by L^1 minimization), we adopt CVXPY [7], an efficient convex optimization tool set, to calculate the L^1 minimum [8]. CVXPY may return no result in some edge cases because the equation is non-homogeneous. Therefore, we gradually increase θ by 0.1 to avoid such a scenario. We set $\theta = 0.2$ based on the evaluation experiments as shown in §4.4. Reconstructing each cluster of univariate time series in a multivariate time series is much more efficient than reconstructing the whole multivariate time series (§4.4).

Anomaly score. We first obtain the reconstructed time series for each cluster of univariate time series to form an original multivariate time series. We then concatenate the reconstructed univariate time series to form a reconstructed multivariate time series \mathbf{X}'_t . Note that the original and reconstructed multivariate time series have the same order of univariate time series. Intuitively, an anomaly score is needed to measure the similarity between the original and the reconstructed multivariate time series. We measure the differences of the n time series between \mathbf{X}_t and \mathbf{X}'_t using euclidean distance [15]: $\mathbf{d}_t^i = |\mathbf{x}_t^i - \mathbf{x}'_t{}^i|$, where $\mathbf{x}'_t{}^i$ is the reconstructed univariate time series of \mathbf{x}_t^i . To avoid an anomaly score being dominated by a single significant spike in a univariate time series, we calculate s_t using the harmonic mean of \mathbf{d}_t^i , i.e., $s_t = n / (\sum_{i=1}^n \mathbf{d}_t^{i-1})$.

Choosing threshold. To properly generate anomaly alerts, we need to accurately choose a threshold to determine whether an anomaly score is high enough to trigger an alert. A static threshold does not work well since the data distribution changes over time. Because an extreme value of the anomaly score generated by *JumpStarter* usually represents an anomaly, we adopt the widely used Extreme Value Theory (EVT) [27] to tailor the anomaly threshold automatically. EVT is a statistical theory aiming to find the law of extreme values, and it does not assume data distribution. It has been demonstrated to accurately choose the threshold for anomaly detection methods [17, 28]. Note that EVT for choosing threshold is not the main contribution of our work.

4 Experiments

In the study, we address the following research questions:

RQ1: How well does *JumpStarter* perform in multivariate time series anomaly detection?

RQ2: Does each component contribute to *JumpStarter*?

RQ3: How do the major parameters of *JumpStarter* influence its performance?

4.1 Experimental Design

4.1.1 Datasets

We conduct experiments on three datasets, including one open dataset¹ – D1 from a large Internet company \mathcal{A} , and two datasets (D2, D3) collected from a top-tier global content platform \mathcal{B} providing services for over 800 million daily active (over 1 billion cumulative) users across all of its content platforms. Specifically, D1 is a five-week-long dataset collected from 28 online service systems, and it is sampled once per minute. These 58 online service systems are located in different servers, which provide services such as searching, ranking, and data processing, etc. D2 and D3 are two datasets

¹<https://github.com/NetManAI/Ops/OmniAnomaly>

Table 3: The detailed information of the datasets (# Training/Test Points = # Online Services * n * # Days * collected data points per day)

Dataset	# Online Services	n	# Training Points (# Days)	# Test Points (# Days)	Anomaly ratio
D1	28	38	19,835,340 (13)	19,835,760 (13)	4.16
D2	30	19	3,283,200 (20)	4,104,000 (25)	5.25
D3	30	19	3,283,200 (20)	4,104,000 (25)	20.26

collected from 30 online service systems over two different seven-week-long periods, respectively. They are both sampled once every five minutes.

This work studies metrics for a single service hosted on one or multiple machines. These metrics are equally important and have no hierarchy among them. The ground truth of anomalies in all the three datasets are manually labeled by operators based on performance issues and failure tickets. The point-wise anomaly rates ($\frac{\# \text{ anomaly data points}}{\# \text{ total data points}}$) are diverse in these datasets. For example, the anomaly rate of D3 (20.26%) is much higher than those of D1 (4.16%) and D2 (5.25%), mainly because D3 contains a severe outage that lasted a long time. Table 3 lists the detailed information of each dataset, including the number of metrics (n), the scale of the training and test sets, and the anomalies ratio. For each service, the monitoring metrics constitute its multivariate time series. The numbers of metrics monitored in D1, D2 and D3 are 38, 19 and 19, respectively. Monitoring tens of metrics is a typical setting for online service systems.

4.1.2 Compared Approaches

We compare *JumpStarter* with two learning-based unsupervised approaches for multivariate time series anomaly detection, namely, MSCRED and OmniAnomaly. We also compare it with two other anomaly detection algorithms: robust random cut forest (RRCF) and least similar nearest neighbors (LESINN). Since it has been demonstrated [28] that univariate time series anomaly detection approaches are not suitable for multivariate time series, we do not compare *JumpStarter* with the baseline methods designed for univariate time series.

RRCF [11]. RRCF is the base anomaly detection algorithm used in AWS CloudWatch, which improves the robustness of original random cut forest probabilistic data structure when detecting anomalies in streaming data. It is open-sourced².

LESINN [20]. LESINN is a time series outlier detection algorithm. For each data point, it calculates the least similar nearest neighbors of it in a time window. If the data point does not have any similar nearest neighbor, it is an outlier.

MSCRED [34]. MSCRED first encodes the temporal correlations among the time series using an attention-based ConvLSTM network, and then reconstructs time series to detect anomalies using a convolutional decoder.

²<https://github.com/aws/random-cut-forest-by-aws>

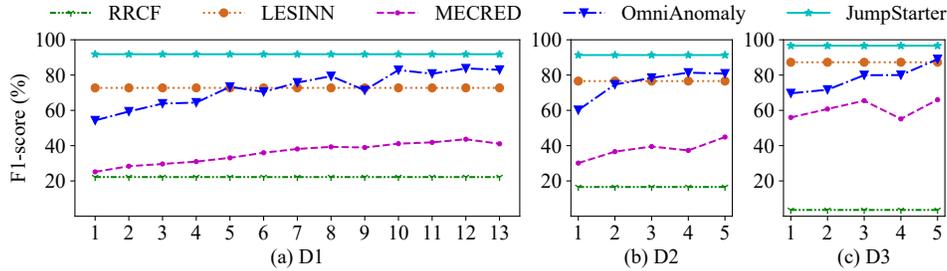


Figure 7: The average F1 score for the three datasets D1, D2, and D3 as a function of the training dataset size in segments.

OmniAnomaly [28]. OmniAnomaly is an unsupervised deep learning based approach. It glues GRU and VAE to model the temporal dependence and stochasticity of time series.

4.1.3 Implementation

JumpStarter is implemented using Python 3.7. For the hyper-parameters, $\rho = 0.1$ and $\sigma = 0.5$ are used in all settings. The detection window size $w = 20$ and the sampling rate $\sigma = 0.2$ are used in §4.4. Our study is conducted on a Dell R420 server with 16 * Intel Xeon E5-2420 CPUs and a 64GB memory.

4.1.4 Evaluation Metrics

The output of a multivariate time series anomaly detection approach for a specific timestamp is either anomalous or not. We use True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN) to label an anomaly detection result according to the ground truth. A TP is an anomaly both confirmed by operators and detected by the approach. If an anomaly is labeled by operators but not detected by the approach, we label the item as an FN. An FP is an “anomaly” that is detected by the approach but is actually normal. An item is TN, if neither the operators nor the approach considers it an anomaly. We use three metrics for evaluating the performance of *JumpStarter* and related approaches: Precision = $TP / (TP + FP)$, Recall = $TP / (TP + FN)$, F1 score = $2 * Precision * Recall / (Precision + Recall)$. The accounting of the three metrics is point-adjusted. That is, if any point in an anomalous segment in the ground truth is detected, we consider the entire segment, or all anomalous points therein, as detected correctly. Point-adjusted metrics are widely adopted in anomaly detection [28, 32], since operators care more about anomalies in a contiguous segment than point-wise anomalies.

4.2 RQ1: Performance of *JumpStarter*

We evaluate two aspects of the performance of anomaly detection each using a different partitioning of training and test sets. First, we conduct service anomaly detection in the online mode and evaluate it as an online experiment. In addition, we collect ten software changes from \mathcal{B} for evaluating the performance of these approaches in reacting to a software

change. Second, in the offline experiment, we adopt the same experiment settings as used in previous work.

Online experiment. We evenly split the training set of D1 into 13 segments (1-day-long data per segment and each has a similar number of anomalies), and D2 and D3 each into 5 segments (4-day-long data per segment and each has a similar number of anomalies). D2 and D3 have a longer segment because they have fewer anomalies per day. For each dataset, the test set remains the same for a fair comparison (see Table 3). Figure 7 shows the average F1 score of *JumpStarter* and four baseline methods as the amount (scale) of training data increases from 1 segment to 13 consecutive segments for D1, and to 5 consecutive segments for D2 and D3. As for *JumpStarter*, RRCF and LESINN, they conduct anomaly detection without any training data. Therefore, their performance stays the same when the scale of training data varies.

We can see that *JumpStarter* performs significantly better than the four baseline approaches across all segments on all the three datasets. RRCF is less accurate than the other approaches because it aims to detect the anomalous behavior of a *single data point*, which is not suitable in our scenario where the anomalous behavior of a *time series segment* is studied. The F1 scores of learning-based approaches, namely OmniAnomaly and MSCRED, increase as the scale of training set increases, and approach 90% and 60% respectively toward the end. OmniAnomaly achieves higher accuracy than LESINN when the amount of training data is sufficient. MSCRED does not perform well because it mainly focuses on the inter-correlations rather than the overall performance of multivariate time series. For all approaches except RRCF, they achieve the best performance on D3 because the anomalous patterns in D3 are easier to capture than those in the other two datasets. The outperformance of *JumpStarter* will be explained next, when we will perform experiments concerning software changes.

Anomaly detection after software changes. Recall that software changes occur frequently in online service systems. We evaluate the performance of the five approaches during ten software changes deployed on two service systems. We do so using the average false positive rate (FPR), defined as $FP / (TN + FP)$, as the metric. FPR measures the burden of false alarms, which is an appropriate metric here since anomalies

Table 4: Average Precision (P), Recall (R), and F1 Score (F) of *JumpStarter* and related approaches

Method	D1			D2			D3			Avg.		
	P	R	F	P	R	F	P	R	F	P	R	F
RRCF [11]	40.26	54.45	39.54	44.06	39.02	30.10	28.33	75.33	38.38	37.55	56.27	36.01
LESINN [20]	75.49	77.40	76.43	77.50	87.15	82.04	87.02	90.95	88.94	80.00	85.17	82.50
MSCRED [34]	46.19	56.16	50.69	46.91	58.26	51.97	68.21	86.47	76.26	53.77	66.96	59.64
OmniAnomaly [28]	78.19	95.03	85.79	77.24	85.84	81.31	89.59	95.02	92.23	81.67	91.96	86.51
<i>JumpStarter</i>	90.35	94.31	92.29	92.05	94.51	93.26	94.14	99.60	96.79	92.18	96.14	94.12

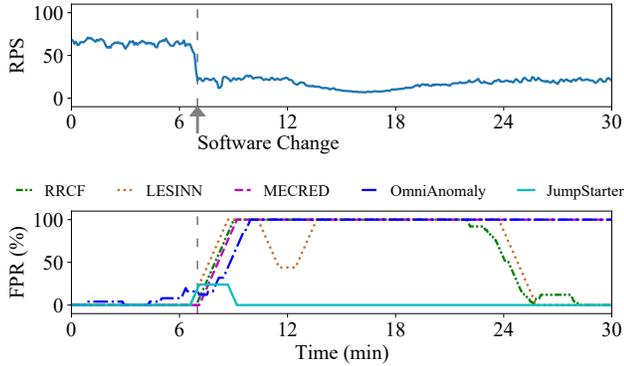


Figure 8: The average False Positive Rate (FPR) during ten software changes. We plot the software changes starting at the seventh minute. The top figure is a constituent univariate time series representing Requests Per Second (RPS).

rarely occur in a short period.

Figure 8 shows the average FPR of the five approaches during the 10 software changes, all of which occur at the seventh minute in the figure. Some constituent univariate times series of the multivariate time series witness level shifts [17]. We observe that all five approaches produce false positives after these software changes. However, *JumpStarter* suffers from high FPR only for about five minutes, after which its FPR becomes quite low. RRCF and LESINN detect anomalies by calculating outliers in a relatively long period, and hence their FPRs do not start to decrease until 17 minutes after software changes. Both OmniAnomaly and MECRED perform well only when the joint distribution of multivariate time series remains unchanged. Consequently, they produce false alarms for a long period after software changes.

Offline experiment. Now we study the potential performance in the offline setting using best F1 score. Since we need a long time to train models of learning-based anomaly detection approaches, we split the dataset into training and test set following the settings in [28]. Then, we calculate the best F1 score of each approach by grid searching their parameters and anomaly thresholds. Table 4 lists the average best F1 scores of *JumpStarter* and baseline approaches on each dataset, as well as their corresponding Precision and Recall. The average best F1 score of *JumpStarter* across the three datasets

Table 5: The average initialization time (IT) and detection time (DT) of *JumpStarter* and baseline approaches

Approach	RRCF	LESINN	MSCRED	Omni-Anomaly	<i>JumpStarter</i>
IT (min)	20	20	>86400	>86400	20
DT (ms)	41.24	118.63	122.82	191.86	127.13

is 94.12%, significantly higher than those of the other four approaches, which are 86.51%, 59.64%, 82.50%, and 36.01%, respectively. This is because D2 contains a large quantity of noises, and none of the other four approaches is robust to such noises. In contrast, *JumpStarter* reconstructs an anomaly-free time series with outlier-resistant sampling, making it robust to such noises in each dataset.

Efficiency. Table 5 lists the average initialization time and detection time of the five approaches. The initialization time of *JumpStarter*, as demonstrated in §4.4, is only twenty minutes, much shorter than those of other approaches. Although RRCF and LESINN achieves the same initialization time as *JumpStarter*, they suffer from low accuracy as shown in Table 4. The detection time of *JumpStarter* for each multivariate time series is 127.13 ms, similar to that of the other approaches. In *JumpStarter*, the shape-based clustering step takes at most 500ms on all three datasets.

4.3 RQ2: Contributions of Components

In this section, we evaluate the relative contributions of two component techniques in *JumpStarter*, namely shape-based clustering and outlier-resistant sampling, to its outperformance. To this end, we reconfigure *JumpStarter* to create three variants. The first two variants concern *JumpStarter* without shape-based clustering. The first variant, called *w/o Clustering: As a Whole*, is to treat all time series as a whole, whereas the second variant, called *w/o Clustering: Separately*, is to sample and reconstruct time series separately. The third variant, called *w/o Sampling*, is *JumpStarter* without outlier-resistant sampling.

Figure 9 shows the comparison results of *JumpStarter* and its three variants on three datasets in terms of average best F1 scores. *JumpStarter* performs better than all its three variants. Specifically, *JumpStarter* improves the average best F1 score of *w/o Clustering: As a Whole*, *w/o Clustering: Separately*, and *w/o Sampling* by 5.81% ~ 14.90%, 2.58% ~ 9.96%, and

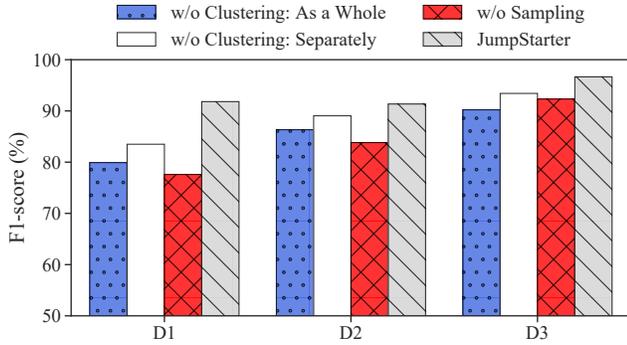


Figure 9: The average F1 score of *JumpStarter* and its three variants on three datasets

4.69% ~ 18.34%, respectively. The results demonstrate that both shape-based clustering and outlier-resistant sampling contribute significantly to a more accurate *JumpStarter*. In addition, *w/o Clustering: Separately* outperforms *w/o Clustering: As a Whole*, because separate reconstruction of time series more accurately captures anomalous patterns than reconstruction as a whole, as shown in Figure 4.

To compare the computational efficiency of different methods, we compare their detection times. For each multivariate time series, the detection time of *w/o Clustering: As a Whole*, *w/o Clustering: Separately*, *w/o Sampling*, and *JumpStarter* is 7891.45 ms, 2056.56 ms, 121.75 ms, 127.13 ms, respectively. This demonstrates that the shape-based clustering technique significantly improves the computational efficiency of *JumpStarter*. In conclusion, the combination of the shape-based clustering and outlier-resistant sampling facilitates an accurate and efficient *JumpStarter*.

4.4 RQ3: Parameter Sensitivity

Recall that our goal is to shorten the initialization time of anomaly detection. The initialization time of *JumpStarter* depends on the detection window size w . We empirically increase the window size from ten minutes to sixty minutes. Figure 10(a) shows how the average best F1 score and point-wise detection time of *JumpStarter* changes as the window size increases. The accuracy of *JumpStarter* increases before the window size reaches 20 minutes, after which it becomes stable, whereas the detection time increases gradually. Therefore, the window size is preset as twenty minutes, which makes *JumpStarter* both accurate and efficient. Note that for those anomalies lasting longer than 20 minutes, *JumpStarter* is still able to detect them because it can easily capture these anomalies when they start.

Another important parameter of *JumpStarter* is σ , the initial sampling rate. Figure 10(b) shows how the average best F1 score and point-wise detection time of *JumpStarter* changes as σ increases. Similarly, when we increase the sampling rate

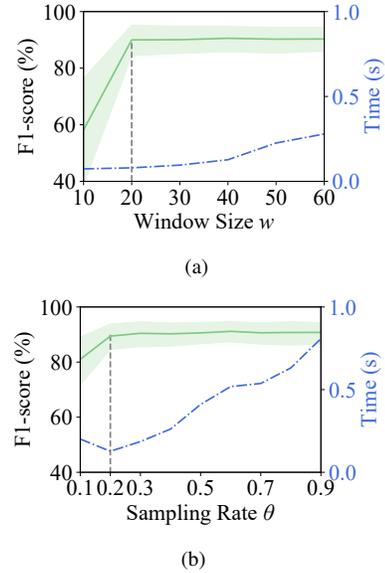


Figure 10: The average F1 score (green line with error bounds) and average point-wise detection time (dotted blue line) of *JumpStarter* under different parameters.

from 0.1 to 0.2, the F1 score of *JumpStarter* increases, and it becomes stable after that. The point-wise detection time of *JumpStarter* decreases with an increase in the sampling rate from 0.1 to 0.2, and it then increases after that. That is because, if the sampling rate is too low (e.g., 0.1), reconstruction is likely to fail, which in turn increases the number of retries. Therefore, we set the sampling rate to 0.2.

5 Deployment and Discussion

5.1 Success Story

Case study. We applied *JumpStarter* into 30 online service systems in a top-tier global content platform \mathcal{B} , which has more than 800 million users around the world. Operators can register a multivariate time series monitor task by ingesting these time series from influxDB and Kafka. From the monitoring dashboard of these services, we can see the multivariate time series of service monitoring. Figure 11 shows some time series of two technical outages.

Case I: Long service response time caused by network issue. As illustrated in Figure 11(a), we observed jitters in time series `tcpext_listdrops` and `tcp_attemptfails`. In the meantime, time series such as `cpu_user` and `load_one` drastically dropped. *JumpStarter* successfully pinpointed this anomaly and generated an alert. After diagnosing the anomaly, operators found that it was a network issue of a database node.

Case II: Service hang-up due to software change. As illustrated in Figure 11(b), time series `tcp_retrans_percentage` witnessed significant jitters and `cpu_idle` plunged to zero. After that, `cpu_sintr`, `cpu_ctxt`, `rx_byptes_eth0` and

tx_pkts_eth0 increased significantly. *JumpStarter* detected and reported this anomaly to operators. Operators conducted software changes and a configuration error occurred in the new version. Thanks to *JumpStarter*, operators found out this error in time and quickly rolled out the software change.

Help with root cause diagnosis. *JumpStarter* can help with root cause diagnosis in two aspects. First, hundreds of multivariate time series need to be monitored. After shape-based clustering in *JumpStarter*, operators can focus on limited time series groups with a similar shape of variations. Second, *JumpStarter* respectively calculates the distance between the original univariate time series and the reconstructed ones. Therefore, it can output a rank list of time series' contributions to the overall anomaly. For example, tcpe_xt_listendrops in Figure 11(a) is detected as the most anomalous time series in this figure. It can explicitly indicate the issue caused by the network component.

5.2 Lessons Learned

Different services may prefer precision and recall differently. With collaboration with different services teams, we found that their preferences on precision and recall are diverse. For example, recall weighs more than precision does in a user interactive core service since operators do not want to miss any potential anomaly that can negatively impact the user experience. In addition, precision is more valuable in a data analysis job because operators would better detect anomalies precisely than to obtain a lot of false alerts. Therefore, the F1 score alone is not a suitable metric for all services. Going forward, we can provide operators with an interface to choose their precision and recall preference level. Specifically, *JumpStarter* can accordingly set the anomaly score using different parameters of the EVT algorithm to guide the sensitivity of detection output. We also observe that severe faults (examples in Figure 11) rarely happen in online services but performance issues do happen a lot. We aim to reduce mean time to restore for severe faults in future work.

Alert system is not just anomaly detection. Our *JumpStarter* for robust and quickly initialized anomaly detection is not the end of the story. Building an intelligent alert system based on anomaly detection results is also a complex task in both engineering and academic aspect. Some anomalies may have no or little signal in the monitored time series. Therefore, *JumpStarter* may miss these anomalies. For this scenario, we aim to collect more types of monitoring data, e.g., logs, traces, to build a more comprehensive anomaly detection model. We believe *JumpStarter* can be easily extended for localizing the anomalous metrics, however, there is a significant gap between anomalous metrics and the root causes of anomalies [16,25]. An intelligent alert system needs to merge similar anomalous cases, pinpoint more sharply to the root causes of anomalies. It also had better learn the priority of differ-

ent anomalous cases [5]. Besides, adeptly integrating domain knowledge into the alert system is also of great importance since the system needs feedback from operators. Therefore, apart from the anomaly detection approach *JumpStarter*, we will improve the alert system behind it.

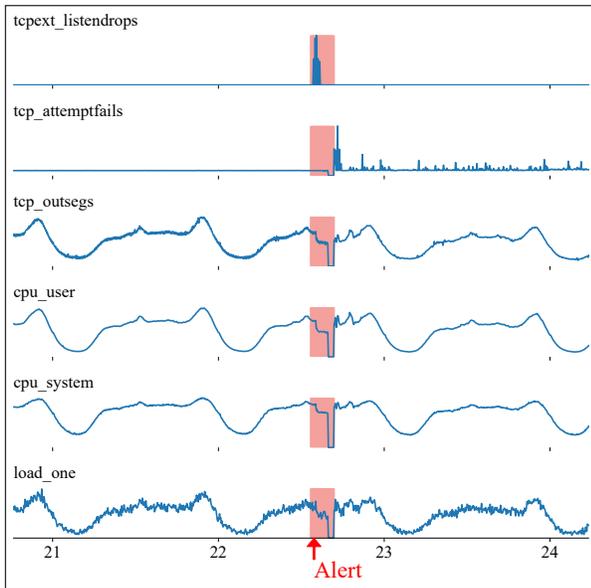
5.3 Threats to Validity

Anomaly labeling. In this work, we use one open dataset from OmniAnomaly and two datasets from real-world services. All the labels in these datasets are provided by operators based on performance issues and incident reports. Manually labeling anomaly points in the timeline may introduce noise (false positives or negatives) because no clear boundaries lie in anomalies and normal patterns. However, domain operators with profound experience suggest the noise in those labels accounts for a very small portion. Besides, operators design evaluation metrics that utilize contiguous anomaly segments instead of point-wise anomalies. Adopting these widely used metrics [17,28,32], we can also eliminate labeling noises.

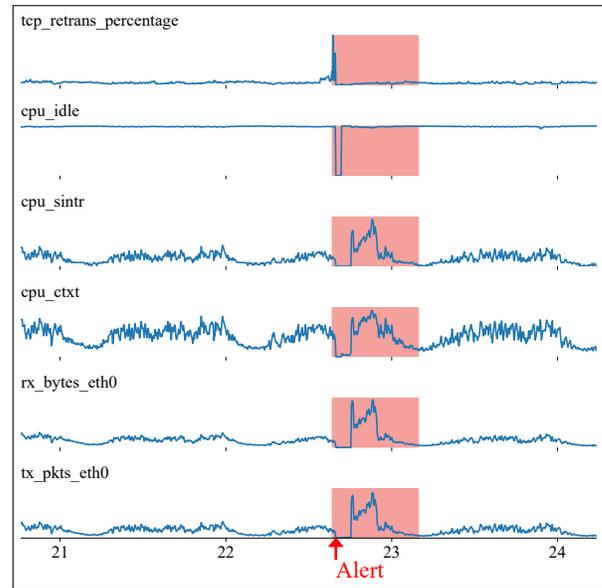
Subject systems. In our experiments, we use an open dataset D1 from a large Internet company. We also collect datasets D2 and D3 from large-scale real-world online service systems. The granularity of the time series of these datasets is one and five minutes, respectively. The efficacy of our algorithm is not influenced by the granularity. With fine-grained granularity, say one second, we believe our algorithm can still work without additional efforts. Since *JumpStarter*'s versatility has been demonstrated using three datasets collected from 58 different services, it should be easy for *JumpStarter* to work with a new dataset. Admittedly, the number of subject services is still limited. We will experiment with *JumpStarter* on a variety of online service systems in the future.

6 Related work

Multivariate time series anomaly detection. Existing approaches include: (1) Traditional statistic method: time series analysis [6], RRCF [11] and clustering-based LESINN [20] do not need training data thus the initialization time is short. However, they all suffer from parameter tuning [15] and being not robust in real practice [17]. (2) Supervised learning based method: Opprentice [15] is an ensemble supervised approach. It needs operators to label anomalies for a long period to train a model. (3) Unsupervised learning based method: LSTM-NDT [12], LSTM-VAE [22], MSCRED [34] and OmniAnomaly [28] build anomaly detection models by learning the anomaly patterns using a large span of historical data. These methods suffer long initialization time for training the model and are less accurate when facing software change [17]. Different from them, we take advantage of compressed sensing, which is a quick initiated signal processing based approach without any training data.



(a) Network Issue



(b) Software Change

Figure 11: Two anomaly cases with selected time series (service performance metrics, *e.g.*, average response time, error rate, are hidden for the confidential reason). Time (X-axis) is shown in days. The alert is generated by *JumpStarter*.

Signal processing based anomaly detection. Since time series of online service systems fluctuate and not always periodic [35]. Other signal processing methods are not suitable in this scenario. For example, Fourier transform can only capture period, the global information of time series [38]. For local information, wavelet analysis can capture local patterns but it is very time consuming [1]. Kalman filtering [18] and PCA [19] are not suitable for variation time series of online service systems [15]. Different from them, our work adopts compressed sensing as the base technique and our experimental results have demonstrated the effectiveness of *JumpStarter* in multivariate time series anomaly detection scenario.

Compressed sensing. As body of theory regarding signal recovery, CS has been widely used in image reconstruction [2], genome-wide association study [30], and many other applications [9]. *JumpStarter* is parallel to them, as it detects anomaly adopting the idea of CS reconstruction.

7 Conclusion

In recent years, online service systems are increasingly deployed on cloud computing platforms. To adapt to frequent changes in online service systems, multivariate time series anomaly detection approaches should be robust and quickly initialized. In this paper, we propose a jump-starting multivariate time series anomaly detection approach with a relatively short initialization time of only twenty minutes. Based on the compressed sensing technique, *JumpStarter* adopts a shape-based clustering strategy to deal with the large number of

univariate time series in a multivariate time series, and outlier-resistant sampling to avoid sampling anomalous values. Experiments conducted on 58 real-world online service systems of two Internet companies demonstrate the effectiveness of *JumpStarter*, achieving an average F1 score of 94.12% and outperforming four state-of-the-art approaches. Besides, our results also endorse the contributions of the main components. In particular, we have applied *JumpStarter* in a real-world online service system and gained some useful lessons.

8 Acknowledgements

We thank our shepherd and the anonymous reviewers for their thorough comments and helpful suggestions. We thank Shiyu Ma, Tiankai Yang, and Hanwen Hu for helping with experiments. We appreciate Christopher Zheng for proofreading our paper. This work has been partially supported by the National Key Research and Development Program of China under Grant No.2019YFE0105500, the National Natural Science of China under Grant 62072264, and the Beijing National Research Center for Information Science and Technology (BNRist) key projects. Shenglin Zhang has been partially supported by the National Natural Science Foundation of China under Grant No. 61902200 and the China Postdoctoral Science Foundation under Grant No.2019M651015. Junjie chen has been partially supported by the National Natural Science Foundation of China under Grand No. 62002256. The work of Jun Xu has been partially supported by US NSF through awards CNS-1909048 and CNS-2007006.

References

- [1] Vicente Alarcon-Aquino and Javier A Barria. Anomaly detection in communication networks using wavelets. *IEE Proceedings-Communications*, 148(6):355–362, 2001.
- [2] Victor J Barranca, Gregor Kovačič, Douglas Zhou, and David Cai. Improved compressive sensing of natural scenes using localized random sampling. *Scientific reports*, 6:31976, 2016.
- [3] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O’Reilly Media, Inc., 2016.
- [4] Emmanuel J Candes and Terence Tao. Decoding by linear programming. *Transactions on Information Theory*, 51(12):4203–4215, 2005.
- [5] Yujun Chen, Xian Yang, Hang Dong, Xiaoting He, Hongyu Zhang, Qingwei Lin, Junjie Chen, Pu Zhao, Yu Kang, Feng Gao, et al. Identifying linked incidents in large-scale online service systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 304–314, 2020.
- [6] David R Choffnes, Fabián E Bustamante, and Zihui Ge. Crowdsourcing service-level network event monitoring. In *Proceedings of the SIGCOMM Conference*, pages 387–398. ACM, 2010.
- [7] Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.
- [8] David L Donoho. For most large underdetermined systems of linear equations the minimal ℓ_1 -norm solution is also the sparsest solution. *Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences*, 59(6):797–829, 2006.
- [9] Yonina C Eldar and Gitta Kutyniok. *Compressed sensing: theory and applications*. Cambridge University Press, 2012.
- [10] João Gama, Indrè Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *Computing Surveys (CSUR)*, 46(4):1–37, 2014.
- [11] Sudipto Guha, Nina Mishra, Gourav Roy, and Okke Schrijvers. Robust random cut forest based anomaly detection on streams. In *International Conference on Machine Learning*, pages 2712–2721. PMLR, 2016.
- [12] Kyle Hundman, Valentino Constantinou, Christopher Laporte, Ian Colwell, and Tom Soderstrom. Detecting spacecraft anomalies using lstms and nonparametric dynamic thresholding. In *Proceedings of the 24th SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 387–395. ACM, 2018.
- [13] Andy C Hung and Teresa H-Y Meng. A comparison of fast inverse discrete cosine transform algorithms. *Multimedia Systems*, 2(5):204–217, 1994.
- [14] Zhihan Li, Youjian Zhao, Rong Liu, and Dan Pei. Robust and rapid clustering of kpis for large-scale anomaly detection. In *Proceedings of the 26th IWQoS International Symposium on Quality of Service*, pages 1–10. IEEE, 2018.
- [15] Dapeng Liu, Youjian Zhao, Haowen Xu, Yongqian Sun, Dan Pei, Jiao Luo, Xiaowei Jing, and Mei Feng. Opprentice: Towards practical and automatic anomaly detection through machine learning. In *Proceedings of the IMC Internet Measurement Conference*, pages 211–224. ACM, 2015.
- [16] Minghua Ma, Zheng Yin, Shenglin Zhang, Sheng Wang, Christopher Zheng, Xinhao Jiang, Hanwen Hu, Cheng Luo, Yilin Li, Nengjun Qiu, et al. Diagnosing root causes of intermittent slow queries in cloud databases. *Proceedings of the VLDB Endowment*, 13(8):1176–1189, 2020.
- [17] Minghua Ma, Shenglin Zhang, Dan Pei, Xin Huang, and Hongwei Dai. Robust and rapid adaption for concept drift in software system anomaly detection. In *Proceedings of the 29th ISSRE International Symposium on Software Reliability Engineering*, pages 13–24. IEEE, 2018.
- [18] Joseph Ndong and Kavé Salamatian. Signal processing-based anomaly detection techniques: a comparative analysis. In *Proc. 2011 3rd International Conference on Evolving Internet*, pages 32–39, 2011.
- [19] Netflix. Rad — outlier detection on big data. <https://netflixtechblog.com/rad-outlier-detection-on-big-data-d6b0494371cc>.
- [20] Guansong Pang, Kai Ming Ting, and David Albrecht. Lesinn: Detecting anomalies by identifying least similar nearest neighbours. In *Proceedings of the ICDMW International Conference on Data Mining Workshop*, pages 623–630. IEEE, 2015.
- [21] John Paparrizos and Luis Gravano. k-shape: Efficient and accurate clustering of time series. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 1855–1870. ACM, 2015.

- [22] Daehyung Park, Yuuna Hoshi, and Charles C Kemp. A multimodal anomaly detector for robot-assisted feeding using an lstm-based variational autoencoder. *Robotics and Automation Letters*, 3(3):1544–1551, 2018.
- [23] Daehyung Park, Hokeun Kim, Yuuna Hoshi, Zackory Erickson, Ariel Kapusta, and Charles C Kemp. A multimodal execution monitor with anomaly classification for robot-assisted feeding. In *Proceedings of the IROS International Conference on Intelligent Robots and Systems*, pages 5406–5413. IEEE, 2017.
- [24] Hansheng Ren, Bixiong Xu, Yujing Wang, Chao Yi, Congrui Huang, Xiaoyu Kou, Tony Xing, Mao Yang, Jie Tong, and Qi Zhang. Time-series anomaly detection service at microsoft. In *Proceedings of the 25th SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 3009–3017. ACM, 2019.
- [25] Zhilei Ren, Changlin Liu, Xusheng Xiao, He Jiang, and Tao Xie. Root cause localization for unreproducible builds via causality analysis over system call tracing. In *Proceedings of the 34th ASE International Conference on Automated Software Engineering*, pages 527–538. IEEE/ACM, 2019.
- [26] Sikandar Samar, Dimitry Gorinevsky, and Stephen Boyd. Likelihood bounds for constrained estimation with uncertainty. In *Proceedings of the 44th Conference on Decision and Control*, pages 5704–5709. IEEE, 2005.
- [27] Alban Siffer, Pierre-Alain Fouque, Alexandre Termier, and Christine Largouet. Anomaly detection in streams with extreme value theory. In *Proceedings of the 23rd SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1067–1075. ACM, 2017.
- [28] Ya Su, Youjian Zhao, Chenhao Niu, Rong Liu, Wei Sun, and Dan Pei. Robust anomaly detection for multivariate time series through stochastic recurrent neural network. In *Proceedings of the 25th SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2828–2837. ACM, 2019.
- [29] Charles F Van Loan. The ubiquitous kronecker product. *Journal of computational and applied mathematics*, 123(1-2):85–100, 2000.
- [30] Shashaank Vattikuti, James J Lee, Christopher C Chang, Stephen DH Hsu, and Carson C Chow. Applying compressed sensing to genome-wide association studies. *GigaScience*, 3(1):2047–217X, 2014.
- [31] Tom Warren. Zoom grows to 300 million meeting participants despite security backlash. <https://www.theverge.com/2020/4/23/21232401/zoom-300-million-users-growth-coronavirus-pandemic-security-privacy-concerns-response>.
- [32] Haowen Xu, Wenxiao Chen, Nengwen Zhao, Zeyan Li, Jiahao Bu, Zhihan Li, Ying Liu, Youjian Zhao, Dan Pei, Yang Feng, et al. Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications. In *Proceedings of the 26th WWW World Wide Web Conference*, pages 187–196, 2018.
- [33] Yong Xu, Kaixin Sui, Randolph Yao, Hongyu Zhang, Qingwei Lin, Yingnong Dang, Peng Li, Keceng Jiang, Wenchi Zhang, Jian-Guang Lou, et al. Improving service availability of cloud systems by predicting disk error. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 481–494, 2018.
- [34] Chuxu Zhang, Dongjin Song, Yuncong Chen, Xinyang Feng, Cristian Lumezanu, Wei Cheng, Jingchao Ni, Bo Zong, Haifeng Chen, and Nitesh V Chawla. A deep neural network for unsupervised anomaly detection and diagnosis in multivariate time series data. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1409–1416, 2019.
- [35] Shenglin Zhang, Ying Liu, Dan Pei, Yu Chen, Xianping Qu, Shimin Tao, and Zhi Zang. Rapid and robust impact assessment of software changes in large internet-based services. In *Proceedings of the 11th CoNext Conference on Emerging Networking Experiments and Technologies*, pages 1–13. ACM, 2015.
- [36] Xu Zhang, Junghyun Kim, Qingwei Lin, Keunhak Lim, Shobhit O Kanaujia, Yong Xu, Kyle Jamieson, Aws Albarghouthi, Si Qin, Michael J Freedman, et al. Cross-dataset time series anomaly detection for cloud systems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1063–1076, 2019.
- [37] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 27th ESEC/FSE Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 807–817. ACM, 2019.
- [38] Nengwen Zhao, Jing Zhu, Yao Wang, Minghua Ma, Wenchi Zhang, Dapeng Liu, Ming Zhang, and Dan Pei. Automatic and generic periodicity adaptation for kpi anomaly detection. *Transactions on Network and Service Management*, 16(3):1170–1183, 2019.

Palleon: A Runtime System for Efficient Video Processing toward Dynamic Class Skew

Boyuan Feng, Yuke Wang, Gushu Li, Yuan Xie, and Yufei Ding
University of California, Santa Barbara
{boyuan,yuke_wang,gushuli,yuanxie,yufeiding}@ucsb.edu

Abstract

On par with the human classification accuracy, convolutional neural networks (CNNs) have fueled the deployment of many video processing systems on cloud-backed mobile platforms (e.g., cell phones and robotics). Nevertheless, these video processing systems often face a tension between intensive energy consumption from CNNs and limited resources on mobile platforms. To address this tension, we propose to accelerate video processing with a widely-available, but not yet well-explored runtime input-level information, namely *class skew*. Through such runtime-profiled information, it strives to automatically optimize CNNs toward the time-varying video stream. Specifically, we build Palleon, a runtime system that dynamically adapts and selects a CNN model with the least energy consumption based on the automatically detected class skews, while still achieving the desired accuracy. Extensive evaluations on state-of-the-art CNNs and real-world videos demonstrate that Palleon enables efficient video processing with up to $6.7\times$ energy saving and $7.9\times$ latency reduction.

1 Introduction

Convolutional neural networks (CNNs) based video processing plays an important role in many emerging applications [3, 4, 7, 9, 10, 16, 31, 35] deployed on cloud-backed mobile platforms. Among them, cognitive assistants and robotic visions are two representative categories. Smart glasses [7, 16, 31], for example, continuously recognize the surrounding environment with CNNs and help the blind person with ordinary tasks (e.g., reading a handwritten note, navigating the grocery store, and even running the Boston Marathon). Robotic visions could automatically search specific animals and document the secret lives of them in the wild [9], as well as detect landmines in various environments [35].

While these applications enjoy both the mobility of the wide deployment in real world and the high accuracy of CNNs, they also face the tension between the limited resource budget on mobile platforms and the high energy consumption and latency of CNNs. A popular CNN, VggNet [67], can easily consume 3.6W and introduce 1.4-second latency, which

makes a large smartphone battery (e.g., 2.7-Ah battery in iPhone X [36]) out of power within 2 hours on continuous image classification. To improve the execution efficiency, many techniques have been proposed such as pruning [28, 46, 48] and quantization [56, 76, 80] to reduce the size of CNN models. However, these existing works fail to exploit the special characteristics of video streams; furthermore, the compromised model accuracy also limits the overall pruning or quantization ratio.

Complementing existing model compression techniques, a strong *temporal locality* in video streams is investigated here to enable efficient video processing on mobile platforms. Considering a video stream collected from a continuous camera feed, it is common that only a small number of classes keep appearing in a large number of consecutive frames. For example, in a film scenario, only a small number of people would come to the master shots frequently, generally lasting for a few minutes, and another group of people will not appear until the scenario has changed. A study on Youtube videos of day-to-day life [65] also shows that more than 90% frames are comprised of less than 10 classes.

We first turn such an abstract concept, *temporal locality*, into something concrete and measurable, *class skew*. Specifically, class skew is formally defined as an unbalanced class distribution that flexibly and effectively extracts the scenario information of both *class cardinality* and *visual separability*. Class cardinality here captures the number of classes in a class skew. By exploiting this class cardinality, we can tailor a general CNN, which is usually trained to recognize thousands of classes, into a specialized model, which only needs to recognize a small number of classes in the current class skew. Meanwhile, we notice that class skews show diverse visual separability under the same class cardinality. For example, a class skew with two classes (e.g., houses and dogs) is easier to recognize compared to that with two more subtle classes (e.g., Husky and Alaskan). By exploiting visual separability, we can use a more compact model for computation and energy saving without loss of accuracy compared with a full model.

We then identify several key challenges that hinder the

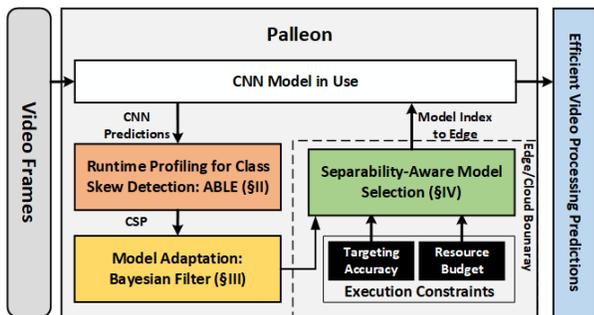


Figure 1: Overview of the Palleon Runtime System.

successful utilization of class skews. First, new class skews may appear and disappear suddenly as time goes, namely *class skew switches*, making it hard to precisely capture the class skew and respond fast to class skew switches. Second, a class skew may last for minutes or even hours between two class skew switches, but this lasting time varies across different videos and even scenarios, thus cannot be decided offline. Third, with the detected class skew, it is still hard to adapt deep models at runtime, since existing model adaptation techniques of retraining fully connected layers are computation-intensive and not affordable on mobile platforms. Forth, a single model adapted toward various class skews may show a significant difference in accuracy due to the diverse visual separability. This difference in accuracy either allows more lightweight CNNs for more energy saving or requires more computation-intensive CNNs to achieve a satisfactory accuracy. Finally, model selection adaptive to class skews may introduce high overhead, making it infeasible to execute on mobile platforms.

To address these challenges, we build a runtime system, Palleon, which could not only detect class skews during runtime, but also dynamically adapt and select a CNN model with the least energy consumption accordingly. In contrast, some existing works [32, 42, 43], which share a similar high-level motivation with us, only target some specific application scenarios that are already known offline—If you want to, you could think of these as "static" class skews without dynamic switches. A recent work, FAST [65], has made some progress along this research direction. FAST assumes that the exact set of class skews (and their duration time) in a video stream are foreknown, and FAST trains a set of compact models for each known class skew offline. During runtime, FAST only needs to detect these foreknown class skews with a simple window-based detector and directly apply those pre-trained models accordingly. To this end, Palleon adopts a pure runtime approach and targets a more realistic setting that the class skews in the video stream are not foreknown.

As illustrated in Figure 1, the Palleon runtime system continuously takes video frames and efficiently generates video processing predictions with three novel components. First, we propose an agile class skew detector, **ABLE** (Section 2), to abstract class skews from video streams. ABLE comes with the *static class-skew profiling* and the *dynamic class-skew*

switch detection. The former automatically detects the class skew and generates a precise *Class Skew Profile* (CSP) when a class skew is detected. The latter continuously catches class skew switches during runtime without the offline information about the class skew lasting time.

Second, we propose **Bayesian Filter** (Section 3) to adapt CNNs toward the detected class skew during runtime. While Bayesian Filter does not directly lead to energy efficiency, it improves the accuracy of compact models with low resource consumption and allows the compact models to replace the complex model. Bayesian Filter is a lightweight module comprised of a *Rescaling* mode that adapts CNNs towards detected CSP without online finetuning and a *Direct Pass* mode that allows the adapted CNNs to still recognize classes out of the current CSP. This lightweight module resolves the complex trade-off between accuracy improvement and adaptation overhead in exploiting class skews.

Third, we design a cloud-backed model selection scheme, namely **Separability-Aware Model Selection** (Section 4), to further squeeze system energy consumption. This scheme exploits visual separability with an *efficient online model selection* that identifies the CNN with the least resource consumption while achieving satisfactory accuracy on the detected class skew. Meanwhile, this scheme contains an *edge-cloud duplicated model bank* to mitigate the model selection overhead on mobile platforms and deliberately schedule the runtime workload between the edge and the cloud.

In summary, we build Palleon, a runtime system that automatically detects input-level information with ABLE and dynamically adapts the given CNNs online with Bayesian Filter. Palleon also controls a set of tuning knobs for balancing the accuracy and the resource efficiency with separability-aware model selection. We build Palleon upon TensorFlow [1] and evaluate it on a cloud-backed mobile platform (with NVIDIA Jetson Nano [40] as the edge device and Dell Workstation T7910 [17] as the cloud server). We evaluate Palleon on various CNN models and different datasets. In particular, for CNN models, we use a variety of the state-of-the-art CNNs from two major domains – object classification (MobileNet [37], VGGNet [67], ResNet [30], and DenseNet [33]) and face recognition (VGGFace [57]). For datasets, we take both synthesized videos and several real-world movies. Extensive experiments confirm the effectiveness of Palleon and show that it could achieve up to $6.7\times$ energy saving and $7.9\times$ latency reduction while achieving an equivalent or better accuracy.

2 ABLE for Class Skew Detection

We build a class-skew detector, namely *ABLE*, to detect class skews during runtime and enable class-skew based optimizations. Our goal is two-fold: 1) giving a precise class-skew profile (CSP) in static regions between adjacent class-skew switches, and 2) detecting when the class-skew switches occur. To this end, we break down our class-skew detection into two sub-tasks: **Static Class-Skew Profiling** and **Dynamic**

Class-Skew Switch Detection.

2.1 Static Class-Skew Profiling

A static CSP generates the distribution of classes in a *static region* where no class skew switch happens. Palleon approximates the CSP in each static region with an empirical distribution [64], which enjoys theoretical properties of converging fast to the ground truth CSP. As illustrated in Figure 2a, given a time window with $r_t = 10$ frames, we collect the predicted labels for each frame and count the frequency of each class. For example, E appears for 4 times out of 10 frames in total, leading to 0.4 for class E in the estimated CSP.

Formally, at time t , in a given frame window with r_t frames (ranging from the $t - r_t + 1^{th}$ to the t^{th} frame), the probability of class j in the CSP is computed as

$$p(j|r_t, x_{1:t}) = \frac{1}{r_t} \sum_{i=t-r_t+1}^t \mathbb{1}_{x_i=j} \quad (1)$$

where x_i is the *predicted label* for the i^{th} frame, $\mathbb{1}_{x_i=j}$ is an indicator function [62] on whether x_i equals j , and $x_{1:t}$ denotes all t predicted labels in the history. The CSP for the given frame window (with r_t frames ending with the t^{th} frame) is computed as a probability vector of all classes:

$$CSP_{t,r_t} = \{p(1|r_t, x_{1:t}), p(2|r_t, x_{1:t}), \dots, p(d|r_t, x_{1:t})\} \quad (2)$$

where d is the total number of classes.

Early Optimization by Adaptive Waiting Scheme. Palleon splits each static region as two phases (Figure 2a): a *waiting phase* to collect a precise CSP based on the full model and an *optimization phase* to apply class-skew based optimizations. In the optimization phase, we use a compact model adapted toward CSPs, which saves energy and reduces latency while achieving an equivalent accuracy to the full model. By allocating smaller number of frames in the waiting phase, Palleon can start the optimization phase early and squeeze more optimization opportunities for more frames, leading to better system performance. However, an imprecise CSP may be generated when allocating too few frames to the waiting phase. Hence, we select the frame number carefully to improve system performance and retain precise CSPs.

We develop an *adaptive waiting scheme* to determine whether Palleon has collected a precise CSP and the waiting phase can be terminated. Suppose the waiting phase has already lasted r_t frames and the current class-skew profile is CSP_{t,r_t} , this scheme computes a minimal frame number F_{min} based on CSP_{t,r_t} . If $F_{min} < r_t$, it terminates the waiting phase. Otherwise, it continues and repeatedly applies such check. In principle, F_{min} guarantees a negligible profiling error ϵ between the true probability p_j and the profiled probability \hat{p}_j

$$\max_{1 \leq j \leq d} |\hat{p}_j - p_j| / p_j \leq \epsilon \quad (3)$$

We next discuss how F_{min} is computed and why it guarantees a negligible profiling error. In addition, we will also

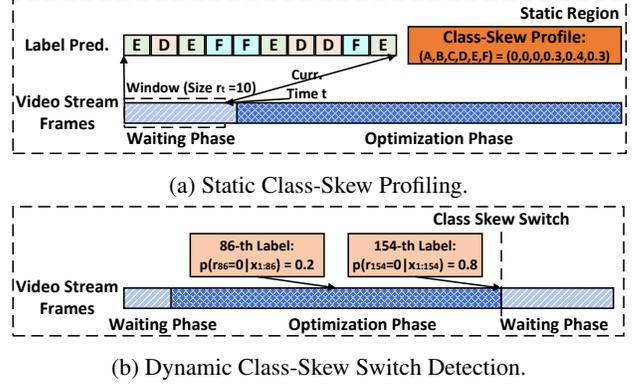


Figure 2: Illustration of ABLE for Class Skew Detection.

propose some practical designs for efficiently computing F_{min} . We start with a theorem, which gives the minimum number of frames to profile a particular class in the class skew.

Theorem 1. (Asymptotic Error Bound). With $n = Z_c / (\epsilon \sqrt{\hat{p}_j})$ samples (frames), the probability of achieving a negligible error $P(|\hat{p}_j - p_j| / p_j < \epsilon) > 1 - c$ for class j holds asymptotically, where Z_c is a Gaussian Distribution Z-score with confidence level $1 - c$ and ϵ is a tolerable error bound.

Proof. Due to the property of multinomial distribution, estimator \hat{p}_j computed with Equation 1 is a maximum likelihood estimator (MLE). Based on the asymptotic normality of MLE, we have $\sqrt{n}(\hat{p}_j - p_j) \rightarrow N(0, FI^{-1})$, where FI is the Fisher Information matrix $FI_{wh} = E_X[-\frac{\partial^2 \ln f_p(x_t)}{\partial p_w \partial p_h}]$. Clearly $FI_{wh} = n/p_w$ if $w = h$; 0, otherwise. Based on the marginalization property of multivariate normality, we can see that $(\hat{p}_j - p_j) \rightarrow N(0, \frac{p_j}{n})$. Following this asymptotic distribution, we can derive the required sample number $n = Z_c / (\epsilon \sqrt{\hat{p}_j})$.

Considering that CSP is stable only if estimators for most classes j are stable, we set the minimum number of frames as

$$F_{min} = \max_{\hat{p}_j > \xi} Z_c / (\epsilon \sqrt{\hat{p}_j(r_t, x_{1:t})}) \quad (4)$$

Here we only consider classes showing *significant existence* ($\hat{p}_j > \xi$, where $\xi = 1/(2 * d)$ is a probability threshold). The intuition is that *CNNs may make wrong predictions randomly spanning in various classes with significantly low probabilities, leading to an unnecessarily long waiting time*. This strategy can mitigate the effect of prediction errors and focus on the effect of correct predictions.

For an arbitrary class number d , computing F_{min} has a low time complexity of $O(d)$, where the main computation resides in iterating through all classes j (Equation 4) and estimating the probability $\hat{p}_j(r_t, x_{1:t})$. This estimation can be conducted efficiently in constant time, based on a *computation reuse* technique detailed in the following section.

2.2 Dynamic Class-Skew Switch Detection

Dynamic class-skew switch detection identifies class skew switches and provides static regions for the static class skew profiling, as shown in Figure 2b. Specifically, dynamic class-

skew switch detection identifies the timestamp t when the previous class skew ends and a new class skew appears. There are two standard techniques to detect class skew switches: Window-based approach [5, 65] and Bayesian-based approach [2, 19, 39, 49, 63, 78]. The former splits video streams into a sequence of windows with a fixed window size k and periodically detects the class skew switch at the boundary. The latter detects class skew switch at each timestamp t by tracking all historical windows, where the $k^{th} \in \{1, 2, \dots, t\}$ historical window contains the $t - k + 1^{th}$ frame to the t^{th} frame. However, the former only detects the class skew switch when a time window has finished, leading to a detection delay up to the fixed window size. While the latter reacts fast to class skew switches by tracking all historical windows, it introduces high overhead with a quadratic time complexity $O((d+t) * t)$, in the number of frames t and the number of classes d . The latter shows more than 1500-millisecond latency per frame after processing a 3-minute video clip. By contrast, Palleon checks class skew switches at each label prediction x_t (Figure 2b) while introducing low computation complexity of $O(d * k)$, where k is the number of sampled windows ($k \ll t$).

At a high level, we sample a subset of window sizes $r_t \in \{w_1, w_2, \dots, w_k\}$ and flag a class skew switch when the probability $p(r_t = 0 | x_{1:t})$ is higher than the probability of the other r_t . We estimate the probability $p(r_t | x_{1:t})$ of each window size r_t when a new predicted label x_t comes:

$$p(r_t | x_{1:t}) = p(r_t, x_{1:t}) / \sum_{r_t=0}^t p(r_t, x_{1:t}), \quad (5)$$

where $p(r_t, x_{1:t})$ is the joint possibility of the lasting time r_t and the predicted labels $x_{1:t}$:

$$p(r_t, x_{1:t}) = \sum_{i=1}^k p(r_t | r_{t-1} = w_i) \cdot p(x_t | r_{t-1} = w_i, x_{1:t-1}) \cdot p(r_{t-1} = w_i, x_{1:t-1}) \quad (6)$$

Here, $p(r_t | r_{t-1} = w_i)$ is a survival function [44] of the probability that a class skew of length r_{t-1} is still alive at r_t , and $p(x_t | r_{t-1} = w_i, x_{1:t-1})$ is the probability that the predicted label x_t comes from the same distribution as last r_{t-1} labels, computed by Equation 1.

Overhead Reduction by Window Sampling. Window sampling selects k windows (*i.e.*, w_1, w_2, \dots, w_k) that minimize the mean absolute error between tracking the selected k windows and all t windows:

$$\min_{w_1, w_2, \dots, w_k} \sum_{r_t=1}^t |p(r_t, x_{1:t}) - p_f(r_t)|, \quad (7)$$

where $f(r_t)$ maps time window r_t to one of the sampled time window $\{w_1, w_2, \dots, w_k\}$. A small number of k windows can approximate all t windows since a window size with low possibility $p(r_{t-1}, x_{1:t-1})$ tends to still have low possibility $p(r_t = r_{t-1} + 1, x_{1:t})$ when new data comes:

$$p(r_t, x_{1:t}) = p(r_t | r_{t-1}) p(x_t | r_{t-1}, x_{1:t-1}) p(r_{t-1}, x_{1:t-1}) \leq p(r_{t-1}, x_{1:t-1}) \quad (8)$$

A straightforward approach is to only track the k most possible windows. However, this approach may not work well when a large number ($> k$) of windows have equally high probability $p(r_t, x_{1:t})$. To this end, we group all t windows into k clusters and select a representative window out of each cluster. To cluster windows, we first sort the possibility $p(r_t, x_{1:t})$ for all time windows r_t and split into clusters at the top $k - 1$ gaps in the sorted sequence. Then, we select the windows with the median probability to represent this cluster and give this window a weight based on the number of windows in the cluster. Intuitively, we exclude the top $k - 1$ gaps by splitting clusters at these gaps to minimize the mean absolute error.

Fast Update by Computation Reuse. Another optimization opportunity is the computation reuse in estimating the conditional distribution $p(x_t | r_{t-1}, x_{1:t-1})$. Since the estimation for each window r_{t-1} at most traverses all data points and all classes, the time complexity of the estimation is linear to the number of data points and classes $O(d + t)$. Since adjacent windows differ only by one input, we can reuse the class frequency in adjacent windows, reducing the time complexity for each window to be $O(d)$. Specifically, the class frequency counts in adjacent windows $r_{t-1} = i$ and $r_{t-1} = i + 1$ differ only by one in a single class, determined by the label x_{t-i} . Thus, with the class frequency count $[C_1, C_2, \dots, C_d]$ in window $r_{t-1} = i$, we can update the frequency count in window size $r_{t-1} = i + 1$ by $C'_j = C_j + 1$, if $j = x_{t-i}$; $= C_j$, o.w.

3 Bayesian Filter for Model Adaptation

In this section, we develop a highly flexible module, namely *Bayesian Filter*, to enable class-skew based optimizations. This module consumes a Class Skew Profile (CSP) detected by ABLE and has two functionalities: 1) adapting CNNs toward the detected class skew with low computation overhead and low latency during runtime; 2) allowing the adapted CNNs to recognize classes out of the current CSP for enabling the detection of class skew switches. To this end, we design a **Rescaling** mode (Section 3.1) for scenario reference (*i.e.*, model adaptation) and a **Direct Pass** mode (Section 3.2) for retaining confident predictions.

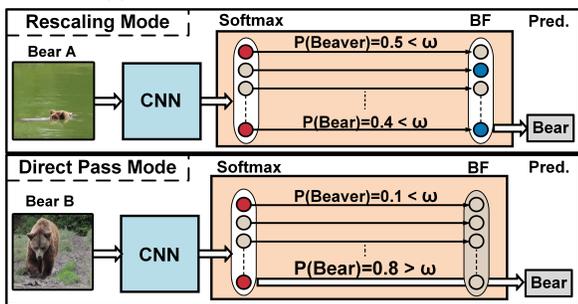
Figure 3a exhibits two videos with extreme class skews, which are intentionally made simple for understanding. In most video frames, the objects are easy to recognize, during which Palleon extracts a precise CSP based on classes recently detected with high confidence. This CSP helps for frames in which objects are hard to recognize (*e.g.*, the animals jump into the water and turn around).

3.1 Rescaling

Rescaling mode is a lightweight module for model adaptation with low computation overhead and low latency. This design avoids the heavy computation overhead and latency in existing work [27, 32, 43, 65], which retrains fully connected layers in CNNs during online and may introduce a long latency (up to 14 seconds) [65]. When considering the energy efficiency,



(a) Two videos with extreme class skews



(b) Intuition behind Bayesian Filter.

Figure 3: Overview of Bayesian Filter.

this retraining procedure either introduces heavy computation overhead when conducted on the edge, or heavy network communication overhead when retraining is conducted on the cloud and updated weights are transferred to the edge. By contrast, Rescaling mode adapts CNNs by appending an extra layer after the fully connected layer. We stress that Rescaling mode requires neither weight update nor model retraining.

Rescaling mode adapts CNNs toward the detected class skew by initializing the extra layer with the CSP generated by ABLE. Since CSP contains the probability of all d classes, the extra layer is designed to have the same number of d nodes, whose weights are initialized by each probability in CSP. In this way, the magnitude of node weights indicates the frequency of the corresponding class in the CSP. When a frame comes, the CNN will generate a probability for each class in the softmax layer and these probabilities will be adjusted according to the magnitude of corresponding node weights. Specifically, the extra layer rescales the probability of softmax-layer predictions (red nodes) toward the current CSP (blue nodes) when the highest softmax-layer probability does not pass a pre-defined threshold ω (*i.e.*, not confident enough). Following the spirit of Bayesian statistics [64], Rescaling mode updates the probability for each class by considering both the prior and the posterior information. We tried several different designs, and it turns out that a simple rescaling scheme based on Bayes theorem would already work effectively, as shown in Formula 9.

$$P(i|X) = \frac{P(i) \cdot P(X|i)}{P(X)}, \quad i \in \{1, 2, \dots, d\} \quad (9)$$

where d is the total number of classes.

Formula 9 computes the posterior probability of class i for a given image X . The prior probability $P(i)$ is the profiled probability of class i in the current CSP. And the likelihood $P(X|i)$ describes the possibility that an image X comes from class i , according to the softmax-layer probability of class i . $P(X)$ stands for the marginal likelihood for observing the image X , which is same for all classes and does not change the rescaling results. Thus, we can avoid computing $P(X)$ and, instead, use a handy rescaling mechanism as $P(i|X) \propto P(i) \cdot P(X|i)$. To the best of our knowledge, we are the first to design Bayesian rescaling on CNNs for runtime model adaptation toward class skews.

3.2 Direct Pass

Direct Pass mode selects the original prediction without rescaling when the predicted probability is higher than a pre-selected threshold ω (Direct Pass mode in Figure 3b). This design allows detecting class skew switches by identifying classes out of the current CSP. This design is inspired by observations [25, 54] that *neural networks usually achieve higher accuracy when they predict a high probability*.

Formally, Bayesian Filter with both Rescaling mode and Direct Pass mode can be written as

$$P(i|X) \propto \begin{cases} P(i) \cdot P(X|i) & \text{if } P(X|i) < \omega \\ P(X|i) & \text{if } P(X|i) \geq \omega \end{cases} \quad (10)$$

where $P(i)$ is the prior probability of observing class i provided by class skew, $P(X|i)$ is the predicted probability from CNNs. We utilize a hyper-parameter ω as the confidence threshold deciding whether Bayesian Filter should enter the direct pass mode. When a model makes a prediction with high probability ($> \omega$), we believe its prediction is correct and Bayesian Filter will not interfere with the decision. Note that two models with different accuracy—for example, a model A with 70% accuracy and a model B with 95% accuracy—could predict with similar high probability when they are making the correct predictions. Their accuracy difference comes from those frames where the poorer model makes a mistake while the stronger model is still correct, not from those frames where both models are correct. Thus, we select the same threshold ω across different CNNs. In particular, we experiment with diverse ω on an extensive collection of the state-of-the-art CNNs and find that a threshold ω between 75% and 95% exhibits similar performance. By default, we use 90% as the threshold ω in following sections.

4 Separability-Aware Model Selection

We propose Separability-Aware Model Selection to enable class-skew based optimizations by exploiting the visual separability. The key observation is that, the same model under different class skew profiles (CSP), even with the same number of classes, may have significantly different accuracy. Figure 4 illustrates two CSPs with different visual separability (*i.e.*,



Figure 4: Class Skews with Different Visual Separability.

one is easy to classify and the other one is hard). To exploit visual separability, Palleon maintains a set of models with different accuracy and energy consumption, and automatically switches to compact models for saving energy when the detected CSP is easy to classify. We are inspired by the fact that people will relax and spend less energy when objects have significantly different appearance, in contrast to distinguishing similar objects (*e.g.*, cat breeds). To this end, we propose an **Efficient Online Model Selection** (Section 4.1) to automatically select models with low resource consumption, and an **Edge-Cloud Duplicated Model Bank** (Section 4.2) to reduce model selection overhead and network overhead.

4.1 Efficient Online Model Selection

We conduct online model selection on the cloud when we detect class skew switches. There are two baseline strategies. One approach records an average accuracy for each model on all classes, and another approach records the accuracy of one model over all possible CSPs (*i.e.*, multiple accuracy for one model). Both approaches are unsatisfactory. On the former approach [27], the selected model may fail to satisfy the accuracy requirement during runtime since the same model may produce significantly different accuracy on different CSPs. On the latter approach [65], a prohibitive offline profiling overhead and online memory overhead may be introduced due to the huge number of CSPs.

By contrast, we propose a hybrid approach that selects models on the cloud for only class skews detected during runtime. During offline preparation, we profile a single accuracy for each model over all classes and store the model in the order of this accuracy. During online model selection, we use binary search to profile the CNN accuracy on the detected CSP. This binary search leads to logarithm time complexity, compared to the linear time complexity of enumerating all models. Behind the binary search, our key observation is that, while the model accuracy on each CSP may change dramatically, the relative accuracy order of models on all classes stays the same over various class skews. In particular, if one model performs better than another model on all classes, the former one generally performs still better than the latter model on various CSPs. Similar observations have also been made in computer vision area [30, 37] that larger models (*e.g.*, ResNet-50) usually give higher accuracy than smaller ones (*e.g.*, ResNet-18) on the same task. Figure 5 illustrates the online model selection. Suppose we have 5 models with decreasing energy consump-

	A	B	C	D	E
CSP I	97	91	87	84	81
CSP II	90	88	82	79	77

Figure 5: Example of Online Model Selection (Unit:%). Dashed boxes refer to un-profiled models due to binary search.

tion and recognition accuracy, and target 90% accuracy. For each CSP, we conduct binary search to find the most compact model with satisfactory accuracy (> 90%). In this case, we will select B for CSP I but A for CSP II.

Cache Service to Avoid Redundant Model Selection.

Palleon records the model selection results along with the CSP and skips model selection for a CSP that have appeared previously. In particular, Palleon maintains a cache service between the CSP and the selected model. When a new CSP comes, Palleon will first retrieve the CSP in the cache. On a cache hit, Palleon immediately returns the selected model. On a cache miss, Palleon conducts online profiling and records the selected model for reuse. In our evaluation, a high cache hit rate is achieved quickly after less than 5 model selections, since the same CSP appears frequently in real videos.

4.2 Edge-Cloud Duplicated Model Bank

Palleon’s goal of saving energy and improving accuracy is affected by the quality of its model bank. We design a duplicated model bank to store only Pareto-Optimal models and duplicate these models on both the edge and the cloud for reducing network overhead. For each candidate model, Palleon stores the computation graph, the pre-trained weights, and the metadata including energy consumption and latency.

Model Bank Generation with Offline Profiling. For each energy budget, we conduct offline profiling to identify candidate models with the highest accuracy. This offline profiling selects only models on the Pareto-optimal curve to reduce online search space and runtime overhead. Specifically, we first generate a large number of candidate models by applying compression techniques on CNNs. Then, we conduct offline profiling to select models on the Pareto-optimal curve [55], defined as the models that we cannot further reduce energy consumption without worsening the accuracy.

This candidate model generation is *conducted once on all classes*, instead of repeating on different CSPs, since good models on all classes tend to consistently produce good performance over various CSPs. The insight is that *unsalient positions remain similar for all CSPs*. For example, when we repeat a compression technique, Perforation [20], for several CSPs on Dense-40 [33], the positions in later blocks will be deleted first while the positions in the leading block remain unchanged until all later blocks have been pruned. More generally, even though the best model (*i.e.*, the one with the highest accuracy under a given reduction in energy consumption) might change between different CSPs, the set of *top-k best* models tends to remain stable over all CSPs. Thus, we can avoid repeating the selection on all CSPs and, under any

Table 1: Profiling on selected compact models. Latency and energy are measured on Jetson Nano.

Compressed From	Layer Remove Ratio (%)	Filter Remove Ratio (%)	Latency (ms)	Energy (J)
ResNet-50	20	10	65.6	0.63
DenseNet-40	30	10	48.4	0.46
MobileNet-128	30	30	34.5	0.35
MobileNet-128	30	40	20.6	0.18
VggNet-19	60	60	10.1	0.07

specific energy-saving requirement, use the best model for all classes to approximate the best model for a specific CSP.

Our full model is a DenseNet-40 with 40 layers. Starting from 4 base models (i.e., MobileNet-128, VGGNet-19, ResNet-50, DenseNet-40), we generate 25 compact models from each of these base models. In particular, we first remove {10%, 20%, 30%, 40%, 60%} of layers from the base model. For the remaining layers, we remove {10%, 20%, 30%, 40%, 60%} of filters. While more sophisticated compression techniques can be applied, we adopt this simple compression technique to validate the effectiveness of model selection. From these pruned models, we select N_{CW} (=5, by default) compact models and put them into our model bank for online use in our evaluation. We have experimented with several numbers and found that 5 compact models can provide a relatively diverse range of accuracy and resource consumption. We show the profiling data on raw latency and raw energy consumption in Table 1, measured on Jetson Nano [40]. For each frame, these models have inference latency from 10.1 ms to 65.6 ms and energy consumption from 0.07J to 0.63J. Here, all pruned models are retrained over all classes during offline model bank generation.

Edge/Cloud Duplication to Reduce Network Overhead.

We maintain a duplicated model bank on both the edge and the cloud to avoid weight transportation from the cloud to the edge. The duplicated model banks on the edge and the cloud contain the same deep models and pre-trained weights, while giving each model an index. During online model selection, the cloud will select a model from the duplicated model bank and only send the selected index to the edge. The edge uses the received index to identify the selected model. This design avoids the network overhead of frequently transporting model weights when class skew switches frequently.

4.3 System Overhead Analysis

Model Bank Memory Overhead. The model bank introduces negligible memory overhead compared to the simple setting with only large CNNs. In the simple setting, the memory consumption is

$$Mem_{Simple} = Mem_{LW} + Mem_{LF} \quad (11)$$

where Mem_{LW} and Mem_{LF} are the memory for storing weights and features of the large CNN, respectively. In our

setting with model bank, the memory consumption is

$$Mem_{Bank} = Mem_{LW} + \max(Mem_{LF}, Mem_{CF}) + N_{CW} \cdot Mem_{CW} \quad (12)$$

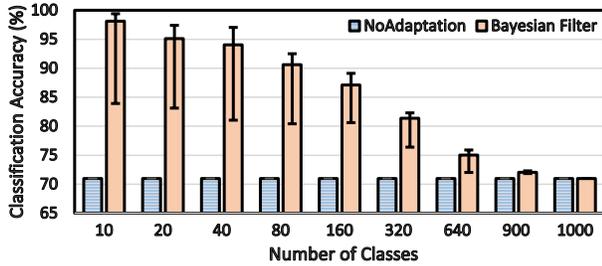
where N_{CW} is the number of compact models, Mem_{CW} and Mem_{CF} are the memory for storing weights and features of compact CNNs. We use $\max(\cdot, \cdot)$ on CNN features since each input frame is processed by only one CNN. Comparing Equation 11 and 12, the model bank only introduces overhead of $N_{CW} \cdot Mem_{CW}$, which is less than 5MB and is negligible compared to the GB-level memory in modern edge devices (GB) (e.g., 1GB in Raspberry Pi 3B+ [59] and 4GB in Jetson Nano [40]). In particular, we use a small N_{CW} (=5, by default), since Bayesian Filter can adapt compact models toward the detected CSPs during runtime with low overhead (Section 3). The Mem_{CW} is usually less than 1MB on compact models generated with compression techniques, especially when the base model also consumes negligible memory (e.g., 0.5MB in SqueezeNet [34] and 2MB in MobileNet [37]).

Runtime Overhead. Palleon’s runtime overhead comes from three sources. The first is the model selection overhead. While this overhead is relatively large, model selection is conducted on the cloud which is powerful and can evaluate several models concurrently. Also, we have sorted the models and proposed a binary search for accelerating the model selection. This procedure introduces negligible runtime overhead (<1%). The second is the data transfer overhead. Existing work usually transfers frames (around 100 KB per frame) to the cloud, which introduces heavy network overhead. Instead, we summarize the surrounding environment into the CSP (a short string within 1KB) and only need to transport the CSP from the edge to the cloud through a wireless network. This network overhead is low since we only transport CSP instead of data or CNN weights. Besides these two overhead from model selection, the third comes from class skew detection on the edge, which is negligible due to optimizations for low-overhead detection in Section 2.2.

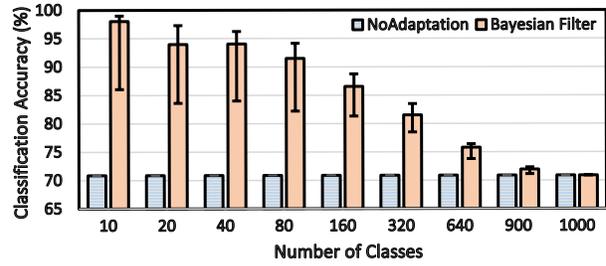
5 Evaluation

To show the effectiveness of Palleon, we perform extensive experiments on both synthesized videos and real videos. We first evaluate Palleon on **synthesized videos** (Section 5.1) to study the performance in diverse settings, including varying class numbers, class types, and lasting time of each class skew. We then conduct **real video experiments** (Section 5.2) to further validate the performance of Palleon for detecting and exploiting class skews during runtime.

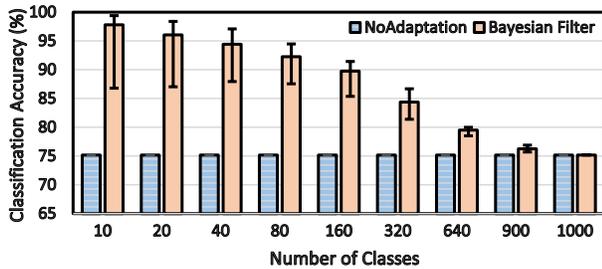
Experiment Platform. We have implemented Palleon in Tensorflow [1] for our CNNs. For the edge device, we use NVIDIA Jetson Nano [40] which is a popular mobile GPU platform with wide deployment in robotics [53], AI glasses [52], and doorbell cameras [51]. Jetson Nano runs Ubuntu 18.04 with built-in support for Tensorflow. For the cloud server, we use a Dell Workstation T7910 with an NVIDIA



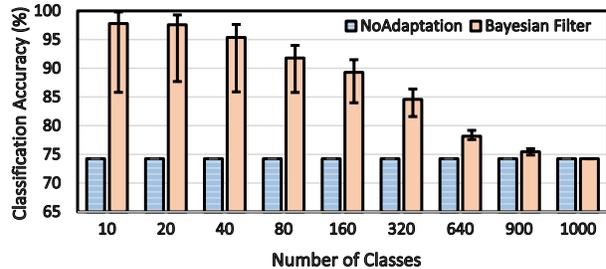
(a) MobileNet with Bayesian Filter.



(b) VGGNet with Bayesian Filter.



(c) ResNet with Bayesian Filter.



(d) DenseNet with Bayesian Filter.

Figure 6: Accuracy on Fixed Class Skews. Error bars represent the accuracy range for each number of classes.

1080Ti GPU (with 11 GB dedicated memory and a nominal peak performance of 11.3 TFLOPS), a 6-core Intel Xeon CPU E5-2603 processor with 32 GB memory running Ubuntu 18.04. All energy measurements mentioned are directly measured unless otherwise specified, using an Extech EX330 Compact Digital Multimeter [50].

5.1 Synthesized Video Experiments

In this section, we extensively evaluate Palleon on synthesized videos in diverse settings. We generate synthesized videos based on ImageNet dataset [18] with diverse class skews. The ImageNet dataset consists of 1,200,000 images categorized into 1,000 classes. We generate class skews with varying numbers of classes and diverse lasting time. These synthesized class skews create challenging scenarios and showcase the robustness of Palleon in challenging settings. We train CNNs on ImageNet with all classes and conduct offline profiling to generate a single accuracy over all classes and collect their latency/energy consumption on mobile devices (e.g., Jetson Nano). This offline profiling is conducted only once. During online, we use Bayesian Filter for online model adaptation and do not use online finetuning (*i.e.*, retraining CNNs on the detected CSPs). On dynamic class skews, we also have online profiling about the model accuracy on the detected CSP, which is conducted on the cloud and introduces negligible overhead.

5.1.1 Bayesian Filter on Fixed Class Skews

Figure 6 shows the accuracy improvement from Bayesian Filter in an ideal case that the true class skew is known and fixed. Under this setting, there is no detection delay and Bayesian Filter can adapt CNNs toward the true class skew. To show the generality of Bayesian Filter, we use four state-of-the-art CNNs as base models (*i.e.*, MobileNet [37], VGGNet [67],

ResNet [30], and DenseNet [33]). When synthesizing fixed class skews, for each $N \in \{10, 20, \dots, 1000\}$ classes, we generate 100 CSPs. Each CSP contains 1000N images by randomly selecting N classes and 1000 images from each class following a uniform distribution. For each number of classes, we run the adapted model and present the average, minimum, and maximum accuracy. We note that we use Bayesian Filter to adapt the model and do not use online finetuning.

Bayesian Filter provides on average 25% accuracy improvement when there are 10 classes. This accuracy improvement shows the effectiveness of Bayesian Filter in adapting models. As the number of classes increases, this accuracy improvement diminishes gradually until all 1,000 classes appear in the class skew (*i.e.*, no scenario information to exploit). The reason is that, as the number of classes increases, opportunities to rule out classes decreases. For example, Bayesian Filter rules out 990 classes when the CSP contains 10 (out of 1000) classes, but only rules out 100 classes when the CSP contains 900 (out of 1000) classes. Surprisingly, an accuracy improvement around 2% still exists when there are 900 classes in the class skew, considering that underlying models can only recognize 1,000 classes. This result shows that Bayesian Filter can consistently improve accuracy on challenging class skews with a large number of classes.

A large accuracy difference exists for each model and each number of classes, demonstrating the existence of visual separability. In particular, an accuracy difference of 15% can be observed for MobileNet when there are 10 classes. This accuracy difference becomes less significant as the number of classes increases, since a smaller number of classes indicates larger variation in the constituent classes. Comparing across models (*e.g.*, ResNet v.s. MobileNet), we see that a model tends to perform better than another model on a specific class

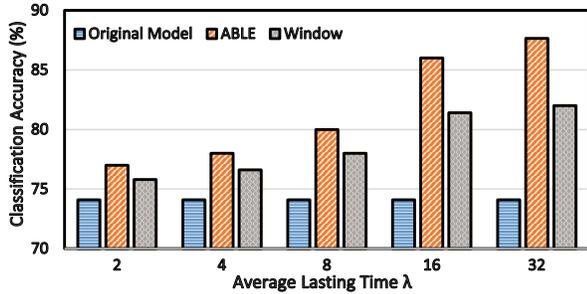


Figure 7: Accuracy with Various Detection Methods.

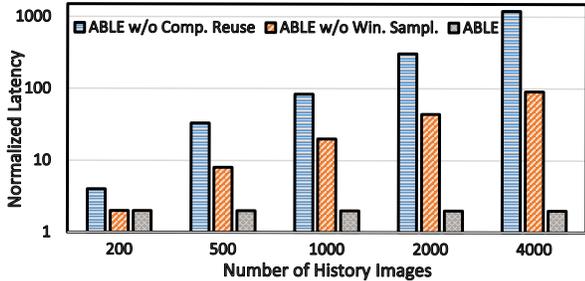


Figure 8: ABLE Latency relative to Window Detector.

skew, if the former model has higher accuracy on all classes than the latter model. This observation indicates that the relative order of model accuracy is invariant over various class skews and supports our binary profiling.

5.1.2 ABLE on Dynamic Class Skews

In this section, we show the accuracy improvement when the true class skew is unknown and may switch abruptly, namely *dynamic class skew*. Under this setting, the class skew detector decides the CSP quality and the detection delay, which has a significant impact on the classification accuracy of the adapted models. When synthesizing dynamic class skews, we randomly generate 30 CSPs. For each CSP, we first randomly select a small number (ranging from 10 to 20) of classes. Among all testing images from a given set of classes, a CSP uniformly samples $lastingTime = 60 * T$ images. T is a random variable following the Poisson(λ) distribution and $\lambda \in \{2, 4, 8, 16, 32\}$ controls the average number of images. We choose the Poisson distribution, as it outputs positive integers. We use DenseNet as the original model and elide the results on other CNNs due to the similar behavior.

Average Accuracy. Figure 7 shows the classification accuracy on dynamic class skews when combining Bayesian Filter with two class skew detectors (*i.e.*, Window and ABLE). In the Window detector, we sample a sequence of window sizes for each synthesized video and present the best accuracy for a strong baseline. Comparing across λ , we can see a clear trend that the classification accuracy increases as λ increases. In particular, “ABLE” can increase accuracy by 13.65% when the average lasting time λ reaches 32. This trend indicates that a class skew lasting longer provides more optimization opportunities to exploit. Comparing across detectors, we can see that ABLE achieves higher accuracy improvement around

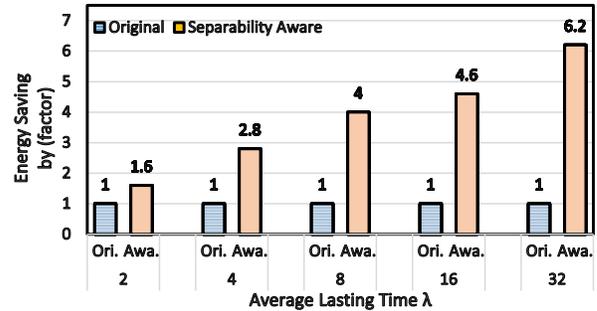


Figure 9: Energy Saving with Model Selection.

5% than the window-based detection. The reason is that the lasting time for a specific class skew varies even for a fixed average lasting time λ , such that a fixed window size can hardly hit the balance between CSP quality and detection delay.

ABLE Detection Latency. Figure 8 shows the ABLE detection latency reduction. This detection latency measures the computation overhead of incurring ABLE on an incoming CNN prediction. *Number of history images* is the total number of images in a synthesized video representing the video length, ranging from 200 to 4000 images. “ABLE” represents ABLE with both computation reuse and window sampling where $k = 30$ windows are sampled. “ABLE w/o Win. Sampl.” disables window sampling and “ABLE w/o Comp. Reuse” further disables computation reuse. “ABLE w/o Comp. Reuse” shows a quadratic increase in the latency over time, which becomes costly when the number of inputs increases gradually. By adding computation reuse, “ABLE w/o Win. Sampl.” decreases this quadratic time complexity to linear time complexity, leading to a much lower computation overhead. When adding window sampling, we can see that “ABLE” further reduces the linear time complexity to a constant complexity, which is similar to the Window detector.

5.1.3 Separability-Aware Model Selection

In this section, we show the energy-saving, runtime speedup, and the memory overhead from Separability-Aware Model Selection. To study the impact of lasting time, we adopt the same setting as dynamic class skew (Section 5.1.2). In each dynamic class skew, we randomly generate 30 class skews and report the average energy saving and runtime speedup.

Energy Saving. Figure 9 shows energy saving when targeting the same accuracy as the baseline model. Palleon can save energy consumption up to $6.2 \times$ while maintaining the accuracy. This benefit comes from automatically replacing the original large model with small models by separability-aware model selection. In addition, we can observe that the energy saving increases as lasting time increases, since a longer lasting time indicates less class skew switches and less system overhead for detecting and exploiting new class skews.

Runtime Speedup. Figure 10 shows the overall runtime speedup when targeting the same accuracy as the baseline model. Palleon can achieve up to $5.48 \times$ speedup, consider-

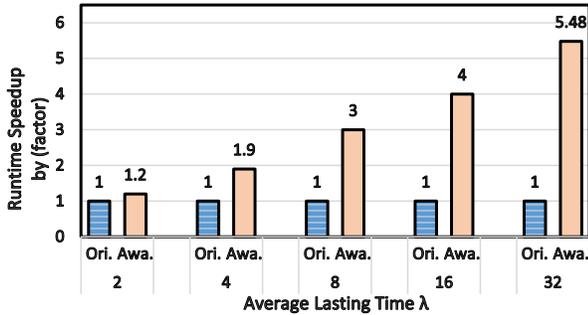


Figure 10: Runtime Speedup with Model Selection.

ing both the model execution speed and the system latency, including the model adaptation latency, the model selection latency on the cloud, and the network latency. Note that the model selection on cloud only introduces latency and has no impact on the energy consumption on the edge device. Similar to the observation in energy saving, we can observe an increase in runtime speedup as the lasting time increases, due to the reduced system overhead.

Memory and data transfer overhead. We observe negligible memory overhead in our current system with 5 compact models. In particular, these compact models usually consume less than 1MB memory and Jetson Nano has 4GB memory. On the data transfer overhead, we only transfer a short CSP (within 1KB) to the cloud when ABLE detects class skew switches. This is significantly smaller than alternative system designs that transfer frames (around 100 KB per frame) or CNN weights with the cloud.

5.2 Real Video Experiments

We evaluate Palleon on real videos to show the end-to-end accuracy improvement, runtime speedup, and energy saving, including the overhead from model adaptation and class skew detection. We compare Palleon with a state-of-the-art energy-efficient video processing system, FAST [65]. FAST approach studies the benefit of class skews in an ideal case, assuming that all class skews that may appear at runtime are known offline. In particular, FAST adapts a large number of compact CNNs towards each class skew during offline preparation and identifies these foreknown class skews with a window detector. As such, we denote it as FAST (offline). For a fair comparison of our online framework, we further extend FAST (offline) to an online version, namely FAST (online). The only difference between these two versions is that FAST (online) does not have the pre-trained CNN models for different class skews. Instead, it adopts a standard retraining method [68], which retrains the last few CNN layers toward the online detected class skews, for online model adaptation. To strike a good balance between accuracy and performance, we manually tune the number of layers for retraining and find that two is a good number and use it in our experiments. Different from FAST, we do not foreknow the class skews offline. We conduct online class skew detection with ABLE, online model adaptation with Bayesian Filter, and online model selection.

Table 2: Real videos for evaluating Palleon. “#Switch” indicates the number of class skew switches and “#Class” shows the average number of classes in each class skew.

Video Name	Len. (min)	#Switch	#Class
Friends	24	45	2.8
Good Will Hunting	14	4	3.5
The Departed	9	8	2.4
Ocean’s Eleven / Twelve	6	25	2.0

Real Video Datasets. We evaluate Palleon and FAST on four real videos [65] depicted in Table 2. These videos come from several movies for face recognition and have diverse length ranging from 6 minutes to 24 minutes. “#Switch” indicates the number of class skew switches in each video and “#Class” represents the average number of classes (faces) in each class skew between adjacent class skew switches. For example, “Friends” is a 24-minute video with 45 class skew switches and each class skew contains 2.8 classes on average. While the total number of classes in these real videos is large (> 20), each class skew contains only a small portion of classes (2 to 3.5). This is a common case in films and Youtube videos, as we have discussed in introduction. The lasting time for each class skew varies on videos from 10 seconds to 4 minutes (about 1.3 minutes on average), computed by “Len.(min) / #Switch”. This diversity makes it a challenging setting to detect and exploit class skews during runtime.

During the detection of faces, we follow the standard two-phase pipeline in object detection [23, 24, 60] and face recognition [6]. We first use a Viola Jones detector [70] to locate faces in video frames, which is agnostic to class skews. Then we crop faces and feed into CNNs for face recognition [57]. Note that this procedure can be easily applied to other object detection tasks by retraining the face recognition CNNs.

Base Model for Real Video Datasets. For a fair comparison with FAST, we choose the state-of-the-art deep model, VGGFace [57], as our full model for face recognition. We generate 5 compact models with diverse resource consumption and accuracy, following the model bank generation in Section 4.2. We use the same compact models for FAST. We train these models from scratch following the hyper-parameter setting in FAST, and achieve comparable accuracy as reported. This offline training is conducted once on LFW dataset. During online, we use Bayesian Filter for model adaptation and do not use online finetuning.

Accuracy Improvement. Figure 11 shows the overall classification accuracy on real videos of the most compact VGGFace model, FAST (offline), and our online approach. We skip the accuracy of FAST (online) since it consistently provides lower accuracy than the Fast (offline) by around 1%, since we retrain only the last two layers in Fast (online) to hit a good balance between accuracy and performance. Palleon provides 6.2% accuracy improvement on average, compared

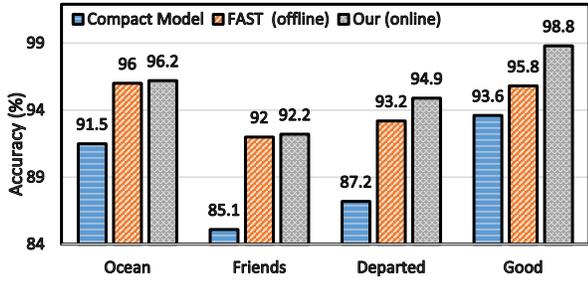


Figure 11: Accuracy Improvement on Various Videos.

to utilizing the compact model without adaptation. This accuracy improvement comes from Bayesian Filter that dynamically adapts models to class skews detected in real videos, containing only 2 to 4 people on average ('#Class' in Table 2). This reduced number of faces greatly eases the task compared to recognizing thousands of faces in un-adapted models.

Comparing to FAST (offline) relying on offline adapted models, Palleon provides 1.3% accuracy improvement due to the faster class skew switch detection in ABLE. When a class skew switches, Window detector in FAST (offline) leads to a detection delay up to 10 seconds, during which the accuracy suffers from a dramatic drop. Moreover, ABLE can effectively detect 98% class skew switches while Window detector can only detect 86% class skew switches, since Window detector fails to detect class skews that exist for only a few seconds.

Runtime Speedup. Figure 12 shows the end-to-end runtime speedup on real videos when targeting the same accuracy as the full model. Palleon achieves on average $5.43\times$ speedup (up to $7.9\times$ speedup on Good) compared to the full model. This speedup comes from automated model selection by replacing the full model with a compact model. When both assuming that the class skews are not foreknown and adapting models at runtime, Palleon achieves $26.9\times$ speedup over the FAST (online) approach. Indeed, FAST (online) shows a $5\times$ slow down due to heavy overhead from online model adaptation. This comparison demonstrates the efficiency of Palleon in online class skew detection and online model adaptation. For FAST (offline) with strong assumption that true class skews are foreknown and models are adapted during offline preparation, Palleon can still achieve a higher speedup, due to the early optimization strategy in ABLE. Comparing across videos, the speedup becomes more significant when class skews have longer lasting time (Good Will Hunting), showing the same pattern as evaluations on synthesized videos (Section 5.1.3). We note that Palleon takes 14.2 ms latency on average to process one frame, which is significantly faster than the real-time requirement of 30 ms per frame.

We also observe negligible overhead from model switches ($<1\%$) due to several reasons. First, the number of model switches is much smaller than the number of class skew switches, since class skew switches can usually be handled by the Bayesian Filter without model switches. In particular, only 20% class skew switches lead to model switches. Second, the

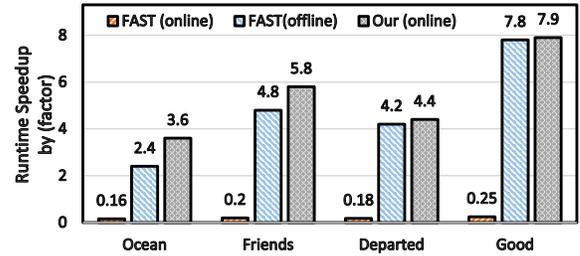


Figure 12: Runtime Speedup on Various Videos.

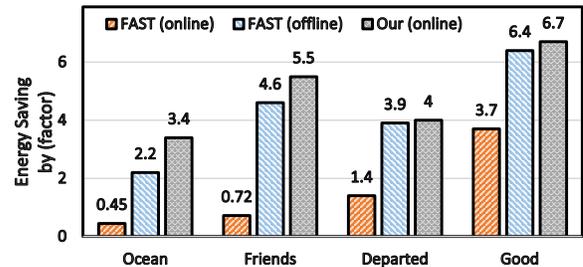


Figure 13: Energy Saving on Various Videos.

model selection is conducted on the cloud and we cache all models in the memory to avoid the repeatedly loading models, which introduces negligible memory overhead.

Energy Saving. Figure 13 shows the end-to-end energy saving on real videos when targeting the same accuracy as the full model. Palleon achieves on average $4.9\times$ energy saving (up to $6.7\times$ on Good) compared to the full model. Palleon achieves a higher energy saving compared to "FAST (offline)" (*i.e.*, without counting the energy consumption in retraining) due to the early optimization strategy in ABLE. FAST (online) conducts model adaptation on the cloud and transfers the adapted model weights (in megabytes) to the edge through the network, leading to extra energy consumption from network communication. This network overhead becomes intensive when class skew switches frequently (Ocean and Friends), leading to more energy consumption in FAST (online) compared to the full model. This overhead reduces when class skews have longer lasting time (Departed and Good), leading to energy saving. By contrast, Palleon shows a consistent benefit on all four videos due to Palleon's low overhead.

Workload Distribution over Models. We observe that most frames are processed by the most compact model. For example, on "Friends" dataset, the most compact model processes 85% frames. While workload distribution varies for different CSPs, we observe similar trends across datasets. The reason is that CSPs usually only contain a few classes (Table 2) and the most compact model with Bayesian Filter can provide high accuracy.

6 Discussion

Comparison with alternative design that trains a model for each CSP. One alternative design to exploit class skews is to train many small models to recognize only a small set of

classes for individual class skews. This alternative design has two intrinsic drawbacks. First, we usually do not foreknow the class skew in an online video such that we can hardly train compact models for each class skew offline. Second, even if we assume that all class skews are foreknown (as the case in FAST), we may need to train a large number of models due to the large number of class skews. By contrast, Palleon does not assume that class skews are foreknown and trains the model on all classes offline. During online video analytics, we use ABLE to conduct online class skew detection, Bayesian Filter for efficient online model adaptation, and separability-aware model selection to automatically select CNNs for balancing the accuracy and resource efficiency.

Generality to other CNNs. Palleon can accelerate a large number of workloads on mobile devices with the temporal locality that a small number of classes keep appearing in a large number of consecutive frames. We have shown the performance benefits of Palleon on object classification and face recognition. Palleon can be generalized to 2D object detection [15] and 3D point cloud analytics [29] which share a similar pipeline as face recognition. We also note that Palleon can benefit from more compact models and pruning techniques designed for mobile systems. In particular, these compact models can be incorporated during model bank generation to provide Pareto-optimal boundary with reduced resource consumption and equivalent accuracy.

7 Related Work

Model Compression. Model compression has been widely explored for accelerating video processing. The popular compression techniques include resolution reduction [21, 37, 45, 58], matrix factorization [38, 61, 77], matrix pruning [14, 28], and distillation [8, 11, 13, 47]. Model compression is orthogonal to our work in exploiting class skews and usually leads to accuracy drop. By contrast, Palleon exploits class skews in video streams and maintains accuracy while reducing energy consumption and processing latency. Meanwhile, Palleon can integrate these compression techniques into our Separability-Aware Model Selection for generating compact models.

Video Processing with Low-Level Temporal Information. Using low-level temporal information can improve accuracy or reduce energy consumption. From the perspective of system design, existing work exploits low-level temporal information by caching processing results of the most recent frames for future computation reuse [26, 74] or adjusting sampling rate [41, 79]. From the perspective of algorithm design, existing work often augments the traditional 2D-CNN with optical flow [66, 71, 72] for explicitly capturing object motions across frames. A new CNN design, 3D-CNNs [12, 22, 69], has also been proposed to implicitly learn object motions by stacking several 2D-CNNs and processing adjacent video frames in a combined way. These works are orthogonal to our work because we focus on exploiting high-level temporal information across minutes, not on low-level temporal infor-

mation in a few seconds. Palleon could be integrated with one of these approaches for further performance improvement.

Video Processing with High-Level Temporal Information. Several video processing systems [27, 32, 43, 65, 73, 75] have been proposed to exploit high-level temporal information across minutes, in terms of scenario information. Several early work [27, 32, 43, 75] simplifies processing tasks by targeting a specific scenario and only recognizing a specific object, *e.g.*, buses at a crosswalk. Recent work [65] conducts offline-profiling over a few scenarios and only reduces energy consumption when these offline-profiled scenarios appear, which would be in-effective for more realistic settings that class skews may switch during runtime. By contrast, Palleon abstracts these specific scenarios to a more general class skew of unbalanced distributions and enables online class skew detection and online model adaptation.

8 Conclusion

Efficient video processing on mobile platforms is an important workload. We present Palleon, a runtime system for efficient video processing, by detecting and exploiting class skews in video streams. We propose ABLE to detect class skews in video streams. Based on these detected class skews, Palleon uses Bayesian Filter for online model adaptation and Separability-Aware Model Selection to select the most energy-efficient model during runtime. Evaluations on both synthesized videos and real videos demonstrate that Palleon achieves up to $6.7\times$ energy saving and up to $7.9\times$ latency reduction. We conclude that Palleon is a highly practical and effective approach for efficiently processing video streams.

9 Acknowledgements

We would like to thank our shepherd, Swami Sundararaman, and the anonymous ATC reviewers. This work was supported in part by NSF 1925717 and 1725447.

References

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, and Michael Isard. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.
- [2] Ryan Prescott Adams and David JC MacKay. Bayesian online changepoint detection. *arXiv preprint arXiv:0710.3742*, 2007.
- [3] Awais Ahmad, Marco Anisetti, Ernesto Damiani, and Gwanggil Jeon. Special issue on real-time image and video processing in mobile embedded systems. *Journal of Real-Time Image Processing*, 16(1):1–4, Feb 2019.

- [4] Advancing ai for video. <https://phys.org/news/2019-06-advancing-ai-video-startup-powerful.html>. Accessed: 2019-11-29.
- [5] Samaneh Aminikhanghahi and Diane J Cook. A survey of methods for time series change point detection. *Knowledge and information systems*, 2017.
- [6] Brandon Amos, Bartosz Ludwiczuk, and Mahadev Satyanarayanan. Openface: A general-purpose face recognition library with mobile applications. Technical report, CMU-CS-16-118, CMU School of Computer Science, 2016.
- [7] Aria. The wearables giving computer vision to the blind. <https://www.wired.com/story/wearables-for-the-blind/>, 2018. Accessed: 2018-07-16.
- [8] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *NeurIPS*, 2014.
- [9] BBC. Bbc series uses robot creatures to document secret lives of animals. <https://www.theguardian.com/media/2016/dec/31/bbc-robot-creatures-spy-secret-lives-animals.-wildlife-series>, 2018. Accessed: 2018-07-16.
- [10] K. M. bin Saipullah, A. Anuar, N. A. binti Ismail, and Y. Soo. Real-time video processing using native programming on android platform. In *2012 IEEE 8th International Colloquium on Signal Processing and its Applications*, pages 276–281, March 2012.
- [11] Cristian Buciluă, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *SIGKDD*, 2006.
- [12] J. Carreira and A. Zisserman. Quo vadis, action recognition? a new model and the kinetics dataset. In *CVPR*, 2017.
- [13] Guobin Chen, Wongun Choi, Xiang Yu, Tony Han, and Manmohan Chandraker. Learning efficient object detection models with knowledge distillation. In *NeurIPS*, 2017.
- [14] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *ICML*, 2015.
- [15] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-FCN: object detection via region-based fully convolutional networks. In *NIPS*, pages 379–387, 2016.
- [16] Dimitrios Dakopoulos and Nikolaos G Bourbakis. Wearable obstacle avoidance electronic travel aids for blind: a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 2009.
- [17] Dell workstation t7910. https://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/Dell_Precision_Tower_7910_Spec_Sheet.pdf. Accessed: 2018-04-20.
- [18] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.
- [19] Paul Fearnhead and Zhen Liu. On-line inference for multiple changepoint problems. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 2007.
- [20] Mikhail Figurnov, Aizhan Ibraimova, Dmitry P Vetrov, and Pushmeet Kohli. Perforatedcnns: Acceleration through elimination of redundant convolutions. In *NeurIPS*, 2016.
- [21] Jianlong Fu, Heliang Zheng, and Tao Mei. Look closer to see better: Recurrent attention convolutional neural network for fine-grained image recognition. In *CVPR*, 2017.
- [22] L. Ge, H. Liang, J. Yuan, and D. Thalmann. 3d convolutional neural networks for efficient and robust hand pose estimation from single depth images. In *CVPR*, 2017.
- [23] Ross Girshick. Fast r-cnn. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV), ICCV '15*, page 1440–1448, USA, 2015. IEEE Computer Society.
- [24] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR '14*, page 580–587, USA, 2014. IEEE Computer Society.
- [25] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. On calibration of modern neural networks. In *ICML*, 2017.
- [26] Peizhen Guo and Wenjun Hu. Potluck: Cross-application approximate deduplication for computation-intensive mobile applications. In *ASPLOS*, pages 271–284, 2018.
- [27] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *MobiSys*, 2016.
- [28] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

- [29] Chenhong He, Hui Zeng, Jianqiang Huang, Xian-Sheng Hua, and Lei Zhang. Structure aware single-stage 3d object detection from point cloud. In *CVPR*, pages 11870–11879. IEEE, 2020.
- [30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.
- [31] Marion Hersch and Michael A Johnson. *Assistive technology for visually impaired and blind people*. Springer Science Business Media, 2010.
- [32] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *OSDI*, 2018.
- [33] Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten. Densely connected convolutional networks. In *CVPR*, volume 1, page 3, 2017.
- [34] 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, 2016.
- [35] Spectrum IEEE. Robot takes on landmine detection while humans stay very very far away. <https://spectrum.ieee.org/automaton/robotics/military-robots/husky-robot-takes-on-landmine-detection-while-humans-stay-very-very-far-away>, 2018. Accessed: 2018-07-16.
- [36] iphone x specification. https://www.gsmarena.com/apple_iphone_x-8858.php. Accessed: 2019-12-3.
- [37] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *CVPR*, 2018.
- [38] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. In *BMVC*, 2014.
- [39] Theodoros Damoulas Jeremias Knoblauch. Spatio-temporal Bayesian on-line changepoint detection with model selection. In *ICML*, 2018.
- [40] Jetson nano specification. <https://developer.nvidia.com/embedded/buy/jetson-nano-devkit>. Accessed: 2018-04-20.
- [41] Angela H. Jiang, Daniel L.-K. Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A. Kozuch, Padmanabhan Pillai, David G. Andersen, and Gregory R. Ganger. Mainstream: Dynamic stem-sharing for multi-tenant video processing. In *ATC*, 2018.
- [42] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: Scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18*, pages 253–266, New York, NY, USA, 2018. ACM.
- [43] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: optimizing neural network queries over video at scale. *VLDB*, 2017.
- [44] David G Kleinbaum and Mitchel Klein. *Survival analysis*, volume 3. Springer, 2010.
- [45] A Krizhevsky and G Hinton. Learning multiple layers of features from tiny images. *Technical report, University of Toronto*, 1, 01 2009.
- [46] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. Runtime neural pruning. In *Advances in Neural Information Processing Systems*, pages 2181–2191, 2017.
- [47] D. Lopez-Paz, B. Schölkopf, L. Bottou, and V. Vapnik. Unifying distillation and privileged information. In *ICLR*, 2016.
- [48] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pages 5058–5066, 2017.
- [49] Othmane Mazhar, Cristian Rojas, Carlo Fischione, and edit Mohammad Reza Hesamzadeh. Bayesian model selection for change point detection and clustering. In *ICML*, 2018.
- [50] Extech multimeter model ex330 manual. http://www.extech.com/resources/EX330_UM.pdf. Accessed: 2018-04-20.
- [51] Build a hardware-based face recognition system for \$150 with the nvidia jetson nano and python. <https://medium.com/@ageitgey/build-a-hardware-based-face-recognition-system-for-150-with-the-nvidia-jetson-nano-and-python-a25cb8c891fd>. Accessed: 2019-06-12.
- [52] Home automation at a glance using ai glasses. <https://hackaday.com/2019/08/15/home-automation-at-a-glance-using-ai-glasses/>. Accessed: 2019-06-12.

- [53] Jetbot, a \$250 diy autonomous robot based on jetson nano impresses at gtc. <https://blogs.nvidia.com/blog/2019/03/26/jetbot-diy-autonomous-robot/>. Accessed: 2019-06-12.
- [54] Alexandru Niculescu-Mizil and Rich Caruana. Predicting good probabilities with supervised learning. In *ICML*, 2005.
- [55] Wikipedia: Pareto efficiency. https://en.wikipedia.org/wiki/Pareto_efficiency. Accessed: 2019-06-12.
- [56] Eunhyeok Park, Junwhan Ahn, and Sungjoo Yoo. Weighted-entropy-based quantization for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5456–5464, 2017.
- [57] Omkar M Parkhi, Andrea Vedaldi, and Andrew Zisserman. Deep face recognition. In *BMVC*, 2015.
- [58] Matthai Philipose. Efficient object detection via adaptive online selection of sensor-array elements. In *AAAI*, 2014.
- [59] Raspberry pi 3b+ specification. <https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/>. Accessed: 2018-04-20.
- [60] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 91–99. Curran Associates, Inc., 2015.
- [61] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. In *ICLR*, 2015.
- [62] W. Rudin. *Principles of Mathematical Analysis*. International series in pure and applied mathematics. McGraw-Hill, 1976.
- [63] Yunus Saatçi, Ryan D Turner, and Carl E Rasmussen. Gaussian process change point models. In *ICML*, 2010.
- [64] J. Shao. *Mathematical Statistics*. Springer Texts in Statistics. Springer, 2003.
- [65] Haichen Shen, Seungyeop Han, Matthai Philipose, and Arvind Krishnamurthy. Fast video classification via adaptive cascading of deep models. In *CVPR*, 2017.
- [66] Zheng Shou, Xudong Lin, Yannis Kalantidis, Laura Sevilla-Lara, Marcus Rohrbach, Shih-Fu Chang, and Zhicheng Yan. Dmc-net: Generating discriminative motion cues for fast compressed video action recognition. In *CVPR*, 2019.
- [67] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.
- [68] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. A survey on deep transfer learning, 2018.
- [69] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. Learning spatiotemporal features with 3d convolutional networks. In *ICCV*, 2015.
- [70] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. *CVPR*, 2001.
- [71] Limin Wang, Yuanjun Xiong, Zhe Wang, Yu Qiao, Dahua Lin, Xiaoou Tang, and Luc Van Gool. Temporal segment networks: Towards good practices for deep action recognition. In *ECCV*, 2016.
- [72] Chao-Yuan Wu, Manzil Zaheer, Hexiang Hu, R Manmatha, Alexander J Smola, and Philipp Krähenbühl. Compressed video action recognition. In *CVPR*, 2018.
- [73] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. A first look at deep learning apps on smartphones. In *WWW*, 2019.
- [74] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. Deepcache: Principled cache for mobile deep vision. In *MobiCom*, 2018.
- [75] Tiantu Xu, Luis Materon Botelho, and Felix Xiaozhu Lin. Vstore: A data store for analytics on large videos. In *EuroSys*, 2019.
- [76] Yuhui Xu, Yongzhuang Wang, Aojun Zhou, Weiyao Lin, and Hongkai Xiong. Deep neural network compression with single and multiple level quantization. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
- [77] Jian Xue, Jinyu Li, Dong Yu, Mike Seltzer, and Yifan Gong. Singular value decomposition based low-footprint speaker adaptation and personalization for deep neural network. In *ICASSP*, 2014.
- [78] Aonan Zhang and John Paisley. Deep Bayesian non-parametric tracking. In *ICML*, 2018.
- [79] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. Live video analytics at scale with approximation and delay-tolerance. In *NSDI*, 2017.
- [80] Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Chris De Sa, and Zhiru Zhang. Improving neural network quantization without retraining using outlier channel splitting. In *International Conference on Machine Learning*, pages 7543–7552, 2019.

FAASNET: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute

Ao Wang¹, Shuai Chang², Huangshi Tian³, Hongqi Wang², Haoran Yang², Huiba Li²,
Rui Du², Yue Cheng¹

¹George Mason University ²Alibaba Group ³Hong Kong University of Science and Technology

Abstract

Serverless computing, or Function-as-a-Service (FaaS), enables a new way of building and scaling applications by allowing users to deploy fine-grained functions while providing fully-managed resource provisioning and auto-scaling. Custom FaaS container support is gaining traction as it enables better control over OSes, versioning, and tooling for modernizing FaaS applications. However, providing rapid container provisioning introduces non-trivial challenges for FaaS providers, since container provisioning is costly, and real-world FaaS workloads exhibit highly dynamic patterns.

In this paper, we design FAASNET, a highly-scalable middleware system for accelerating FaaS container provisioning. FAASNET is driven by the workload and infrastructure requirements of the FaaS platform at one of the world's largest cloud providers, Alibaba Cloud Function Compute. FAASNET enables scalable container provisioning via a lightweight, adaptive function tree (FT) structure. FAASNET uses an I/O efficient, on-demand fetching mechanism to further reduce provisioning costs at scale. We implement and integrate FAASNET in Alibaba Cloud Function Compute. Evaluation results show that FAASNET: (1) finishes provisioning 2,500 function containers on 1,000 virtual machines in 8.3 seconds, (2) scales 13.4× and 16.3× faster than Alibaba Cloud's current FaaS platform and a state-of-the-art P2P container registry (Kraken), respectively, and (3) sustains a bursty workload using 75.2% less time than an optimized baseline.

1 Introduction

In recent years, a new cloud computing model called serverless computing or Function-as-a-Service (FaaS) [40] has emerged. Serverless computing enables a new way of building and scaling applications and services by allowing developers to break traditionally monolithic server-based applications into finer-grained cloud functions. Developers write function logic while the service provider performs the notoriously tedious tasks of provisioning, scaling, and managing the backend servers [36] that the functions run on.

Serverless computing solutions are growing in popularity and finding their way into both commercial clouds (e.g., AWS Lambda [5], Azure Functions [7], Google Cloud Functions [12] and Alibaba Cloud Function Compute¹ [2], etc.) and open source projects (e.g., OpenWhisk [54], Knative [15]). While serverless platforms such as AWS Lambda

and Google Cloud Functions support functions packaged as .zip archives [8], this deployment method poses constraints for FaaS applications with a lack of flexibility. One constraint is a maximum package size limit (of up to 250 MB uncompressed for AWS Lambda functions).

A recent trend is the support of packaging and deploying cloud functions using custom container images [3, 13, 19, 20]. This approach is desirable as it greatly enhances usability, portability, and tooling support: (1) Allowing cloud functions to be deployed as custom container runtimes enables many interesting application scenarios [4], which heavily rely on large dependencies such as machine learning [23, 47], data analytics [31, 32, 58], and video processing [26, 35]; this would not have been possible with limited function package sizes. (2) Container tooling (e.g., Docker [9]) simplifies the software development and testing procedures; therefore, developers who are familiar with container tools can easily build and deploy FaaS applications using the same approach. (3) This approach will enable new DevOps features such as incremental update (similar to rolling update in Kubernetes [18]) for FaaS application development.

A potential benefit that makes the FaaS model appealing is the fundamental resource elasticity—ideally, a FaaS platform must allow a user application to scale up to tens of thousands of cloud functions on demand, in seconds, with no advance notice. However, providing rapid container provisioning for custom-container-based FaaS infrastructure introduces non-trivial challenges.

First, FaaS workloads exhibit highly dynamic, bursty patterns [50]. To verify this, we analyzed a FaaS workload from a production serverless computing platform managed by one of the world's largest cloud providers, *Alibaba Cloud*. We observe that a single application's function request throughput (in terms of concurrent invocation requests per second or RPS) can spike up to more than a thousand RPS with a peak-to-trough ratio of more than 500× (§2.2.1). A FaaS platform typically launches many virtualized environments—in our case at Function Compute, virtual machines (VMs) that host and isolate containerized functions—on demand to serve request surges [5, 24, 56]. The bursty workload will create a network bandwidth bottleneck when hundreds of VMs that host the cloud functions are pulling the same container images from the backing store (a container registry or an object store). As a result, the high cost of the container startup process²

¹We call Function Compute throughout the paper.

²A container startup process typically includes downloading the container

makes it extremely difficult for FaaS providers to deliver the promise of high elasticity.

Second, custom container images are large in sizes. For example, more than 10% of the containers in Docker Hub are larger than 1.3 GB [60]. Pulling large container images from the backing store would incur significant cold startup latency, which can be up to several minutes (§2.2.2) if the backing store is under high contention.

Existing solutions cannot be directly applied to our FaaS platform. Solutions such as Kraken [16], DADI [42], and Dragonfly [10] use peer-to-peer (P2P) approaches to accelerate container provisioning at scale; however, they require one or multiple dedicated, powerful servers serving as root nodes for data seeding, metadata management, and coordination. Directly applying these P2P-based approaches to our existing FaaS infrastructure is not an ideal solution due to the following reasons. (1) It would require extra, dedicated, centralized components, thus increasing the total cost of ownership (TCO) for the provider while introducing a performance bottleneck. (2) Our FaaS infrastructure uses a dynamic pool of resource-constrained VMs to host containerized cloud functions for strong isolation; a host VM may join and leave the pool at any time. This dynamicity requires a highly adaptive solution, which existing solutions fail to support.

To address these challenges, we present FAASNET, a lightweight and adaptive middleware system for accelerating serverless container provisioning. FAASNET enables high scalability by decentralizing the container provisioning process across host VMs that are organized in function-based tree structures. A *function tree* (FT) is a logical, tree-based network overlay. A FT consists of multiple host VMs and allows provisioning of container runtimes or code packages to be decentralized across all VMs in a scalable manner. FAASNET enables high adaptivity via a tree balancing algorithm that dynamically adapts the FT topology in order to accommodate VM joining and leaving.

Note that the design of FAASNET is driven by the specific workload and infrastructure requirements of Alibaba Cloud Function Compute. For example, Function Compute uses containers inside VMs to provide strong tenant-level isolation. A typical FaaS VM pool has thousands of small VM instances. The scale of the FaaS VM pool and the unique characteristics of FaaS workloads determine that: (1) a centralized container storage would not scale; and (2) existing container distribution techniques may not work well in our environment as they have different assumptions on both workload types and underlying cluster resource configurations.

We make the following contributions in this paper.

- We present the design of a FaaS-optimized, custom container provisioning middleware system called FAASNET. At FAASNET’s core is an adaptive function tree abstraction that avoids central bottlenecks.

image manifest and layer data, extracting layers, and starting the container runtime; in our paper we call the startup process *container provisioning*.

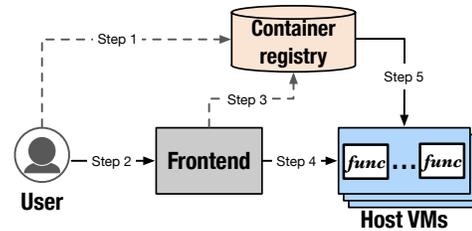


Figure 1: Overview of Alibaba Cloud’s FaaS container workflows.

- We implement and integrate FAASNET in Alibaba Cloud Function Compute. FAASNET is, to the best of our knowledge, the first FaaS container provisioning system from a cloud provider with published technical details.
- We deploy FAASNET in Alibaba Cloud Function Compute and evaluate FAASNET extensively using both production workloads and microbenchmarks. Experimental results show that FAASNET: finishes provisioning 2,500 function containers within 8.3 seconds (only $1.6\times$ longer than that of provisioning a single container), scales $13.4\times$ and $16.3\times$ faster than Alibaba Cloud’s current FaaS platform and a state-of-the-art P2P registry (Kraken), respectively, and sustains a bursty workload using 75.2% less time than an optimized baseline.
- We release FAASNET’s FT and an anonymized dataset containing production FaaS cold start traces at <https://github.com/mason-leap-lab/FaaSNet>.

2 Background and Motivation

In this section, we first provide an overview of the FaaS container workflows in Alibaba Cloud Function Compute. We then present a motivational study on the FaaS workloads to highlight the bursty patterns and their demands of a scalable and elastic FaaS container runtime provisioning system.

2.1 FaaS Container Workflows in Alibaba Cloud Function Compute

Function Compute allows users to build and deploy FaaS applications using custom container images and container tools. Figure 1 shows a typical workflow of function deployment and invocation. To deploy (or update) a containerized function, a user sends a `create/update` request in order to push the container image to a centralized container registry (Step 1 in Figure 1). To invoke a deployed function, the user sends `invoke` requests to the frontend gateway (Step 2), which checks the user’s request and the status of the container image in the registry (Step 3). The frontend then forwards the requests to the backend FaaS VM cluster for servicing the requests (Step 4). Finally, host VMs create function containers and pull their container data from the registry (Step 5). After all the previous steps are successfully completed, host VMs become ready and start serving the invocation requests.

2.2 Workload Analysis

Step 5 in Figure 1, container runtime provisioning, must be fast and scalable in order to enable high elasticity for

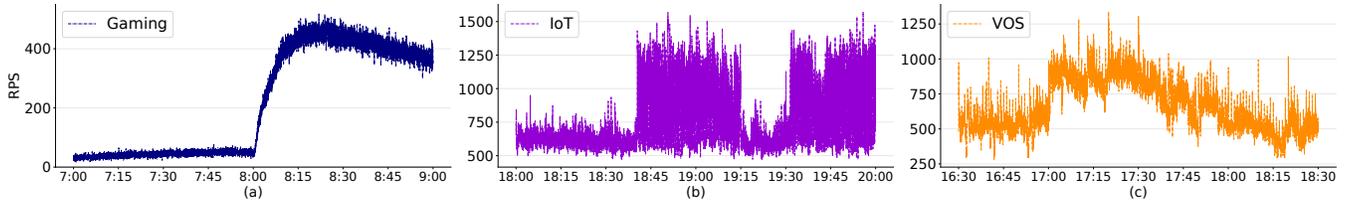


Figure 2: 2-hour throughput timelines of example FaaS applications.

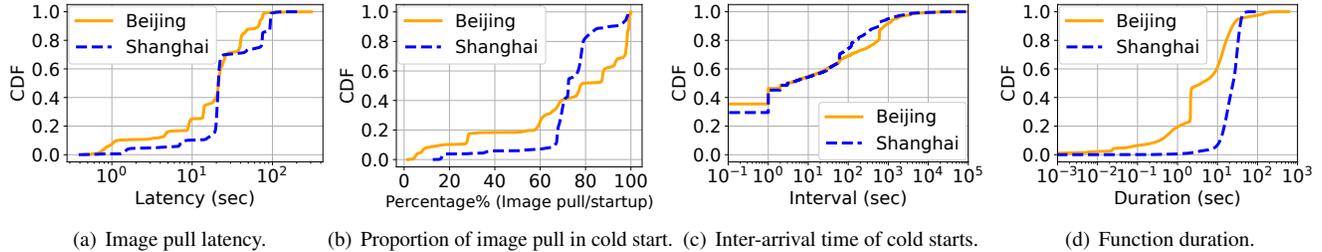


Figure 3: Performance characteristics of container image pulls (a, b) and function invocations (c, d) in CDF.

FaaS workloads. To obtain a better understanding of the workload requirements, we analyze the workload traces that were collected from Function Compute.

2.2.1 Workload Burstiness

FaaS providers charge users using a fine-grained, pay-per-use pricing model—they bill on a per invocation basis (e.g., \$0.02 per 1 million invocations for AWS Lambda) and charge the CPU and memory bundle resource usage at the millisecond level. This property is attractive to a broad class of applications that exhibit highly fluctuating and sometimes unpredictable loads; compared to traditional VM-based deployment approach that charges even when the VM resources are idle, FaaS is more cost-effective as tenants do not pay when the load is zero. Therefore, we analyzed the workload traces and verified that bursty behaviors are indeed common. Figure 2 reports the behaviors of three representative FaaS applications: gaming, IoT, and VOS (video processing).

Figure 2(a) shows that a request spike shoots from 22 to 485 RPS with a peak-to-trough ratio of $22\times$. As well as being bursty, IoT and VOS show different patterns. As shown in Figure 2(b), IoT exhibit a sustained throughput of around 682 RPS, but the throughput suddenly increased to more than 1460 RPS; the peak throughput lasts for about 40 minutes and the second peak starts 15 minutes after the first peak ends. Whereas for VOS (Figure 2(c)), for the first 30 minutes, it observes an average throughput of 580 RPS with a maximum (minimum) throughput of 982 (380) RPS; the average throughput increases to 920 RPS at 30 minutes, and gradually reduces back to an average of 560 RPS.

Implication 1: *Such dynamic behaviors require scalable and resilient provisioning of large numbers of function containers to rapidly smooth out the latency spikes that a FaaS application may experience during a request burst.*

2.2.2 Cold Start Costs of Containerized Functions

Next, we focus on cold start costs of containerized functions. A cold start, in our context, refers to the first-ever invocation of a custom-container-based function; a cold start latency is typically long, ranging from a few seconds to a few minutes as it requires the FaaS provider to fetch the image data and start the container runtime before executing the function. As noted in prior work [25, 48, 50, 56], the high cold start penalty is a notorious roadblock to FaaS providers as it hurts elasticity. The cold start issue is exacerbated when custom container feature with sizeable dependencies is supported.

We analyzed the container downloading costs in two FaaS regions, *Beijing* and *Shanghai*, managed by Function Compute. We retrieved a 15-day log, which recorded the statistics of the function container registry and reported the performance characteristics of 712,295 cold start operations for containerized functions. As shown in Figure 3(a), for *Beijing*, about 57% of the image pulls see a latency longer than 45 seconds, while for *Shanghai* more than 86% of the image pulls take at least 80 seconds.

We next examined the proportion of time spent on image pull with respect to the total function cold start latency. Figure 3(b) shows that more than 50% and 60% of function invocation requests spend at least 80% and 72% of the overall function startup time on pulling container images, for *Beijing* and *Shanghai* respectively. This indicates that the cost of image pull dominates most functions' cold start costs.

To put the cold start costs into perspective, we further inspected cold starts' inter-arrival time and function duration. Figure 3(c) plots the interval distribution of consecutive cold start requests. In both of the two regions, about 49% of function cold starts have an inter-arrival time less than 1 second, implying a high frequency of cold start requests. As shown in Figure 3(d), about 80% of the function executions in *Beijing* region are longer than 1 second; in *Shanghai* region, about

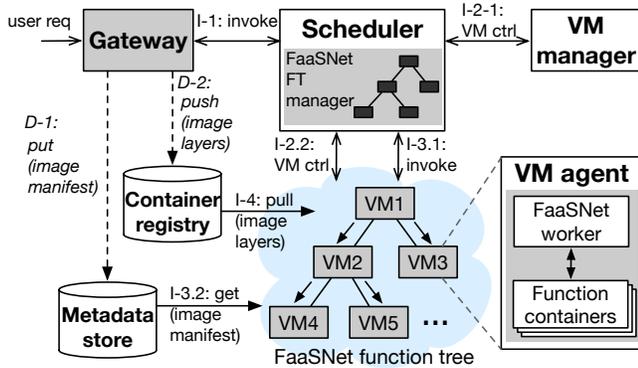


Figure 4: FAASNET architecture. Our work is in the gray boxes. Function invocation requests (solid arrow: invoke, VM ctrl, get(image manifest), and pull(image layers)) are online operations. FAASNET minimizes the operation of I-4: pull(image layers) and efficiently decentralizes container provisioning (i.e., image load and container start) across VMs. Function deployment requests (dashed arrow, italic font: *D-1: put(image manifest)* and *D-2: push(image layers)*) are offline operations.

80% of the function duration is less than 32.5 seconds, with a 90th percentile of 36.6 seconds and a 99th percentile of 45.6 seconds. This distribution indicates that cold start costs are of the same magnitude as the function duration, stressing a need for optimizing container startups.

Implication 2: *Optimizing the performance of container provisioning will provide a huge benefit on reducing the cold start costs of container-based cloud functions.*

3 FAASNET Design

3.1 Design Overview

This section provides a high-level overview of FAASNET’s architecture. Figure 4 illustrates the architecture of the FaaS platform running FAASNET. FAASNET decentralizes and parallelizes container provisioning³ across VMs. FAASNET introduces an abstraction called *function trees (FTs)* to enable efficient container provisioning at scale. FAASNET integrates a *FT manager* component and a *worker* component into our existing FaaS scheduler and VM agent for coordinated FT management. Next, we describe the main components in our FaaS platform.

A *gateway* is responsible for (1) tenant identity access management (IAM) authentication, (2) forwarding the function invocation requests to the FaaS scheduler, and (3) converting regular container images to the I/O efficient data format.

A *scheduler* is responsible for serving function invocation requests. We integrate a FAASNET *FT manager* into the scheduler to manage *function trees* (§3.2), or *FTs* for short, through *FT’s insert* and *delete* APIs. A *FT* is a binary tree overlay that connects multiple host VMs to form a fast and

³While this paper mainly focuses on container runtime provisioning, FAASNET supports provisioning of both containers and code packages.

scalable container provisioning network. Each VM runs a FaaS VM agent, which is responsible for VM-local function management. We integrate a FAASNET *worker* into the VM agent for container provisioning tasks.

On the function invocation path, the scheduler first communicates with a *VM manager* to scale out the the active VM pool from a free VM pool, if there are not enough VMs or all VMs that hold an instance of the requested function are busy. The scheduler then queries its local FT metadata and sends RPC requests to FAASNET workers of the FT to start the container provisioning process (§3.3). The container runtime provisioning process is effectively decentralized and parallelized across all VMs in a FT that do not yet have a container runtime locally provisioned. The scheduler sits off the critical path while FAASNET workers fetch function container layers on demand and creates the container runtime (§3.5) from the assigned peer VMs in parallel.

As described in §2.1, on the function deployment path, the gateway converts a function’s regular container image into an *I/O efficient* format (§3.5) by pulling the regular image from a tenant-facing container registry, compresses the image layers block-by-block, creates a metadata file (an image manifest) that contains the format-related information, and writes the converted layers and its associated manifest to an Alibaba Cloud-internal container registry and a metadata store, respectively.

3.2 Function Trees

We make the following design choices when designing FTs. (1) A function has a separate FT; that is, FAASNET manages FTs at function granularity. (2) FTs have decoupled data plane and control plane; that is, each VM worker in a FT has equivalent, simple role of container provisioning (data plane), and the global tree management (control plane) to the scheduler (§3.3). (3) FAASNET adopts a balanced binary tree structure that can dynamically adapt to workloads.

These design choices are well aligned with Alibaba Cloud’s existing FaaS infrastructure and are attuned to achieve three goals: (1) minimizes the I/O load of container image and layer data downloading on backing container registry, (2) eliminates the tree management bottleneck and data seeding bottleneck of a central root node, and (3) adapts when VMs join and leave dynamically.

Managing Trees at Function Granularity. FAASNET manages a separate, unique tree for each function that has been invoked at least once and has not been reclaimed. Figure 5 illustrates the topology of a three-level FT that spans five host VMs. Function container images are streamed from the root VM of the tree downwards until reaching the leaf nodes.

Balanced Binary Trees. At FAASNET’s core is a balanced binary tree. In a binary tree, except for the root node and leaf nodes, each tree node (in our case a host VM)⁴ has one incoming edge and two outgoing edges. This design can

⁴We use “node”/“VM” interchangeably when describing tree operations.

effectively limit the number of concurrent downloading operations per VM to avoid a network contention. A balanced binary tree with N nodes has a height of $\lfloor \log(N) \rfloor$. This is desirable as a balanced binary tree guarantees that the image and layer data of a function container would traverse at most $\lfloor \log(N) \rfloor$ hops from the top to the bottom. This is critical as the height of a FT would affect the efficiency of data propagation. Furthermore, the structure of a balanced binary tree can dynamically change in order to accommodate the dynamicity of the workloads. To this end, FAASNET organizes each FT as a balanced binary tree. The *FT manager* (Figure 4) calls two APIs, *insert* and *delete*, to dynamically grow or shrink a FT.

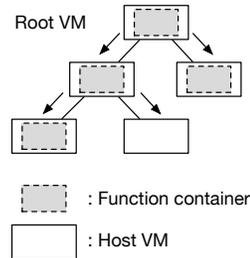


Figure 5: An example FAASNET FT.

insert: The very first node of a FT is inserted as a root node. The FT manager tracks the number of child nodes that each tree node has via BFS (breadth-first search) and stores all nodes that has 0 or 1 child in a queue. To insert a new node, the FT manager picks the first node from the queue as the parent of the new node.

delete: The scheduler may reclaim a VM that has been idling for a period of time (15-minute in Alibaba Cloud configuration). Thus, FaaS VMs have a limited lifespan. To accommodate VM leaving caused by reclamation, the FT manager calls *delete* to delete a reclaimed VM. The *delete* operation rebalances the structure of FT if needed. Different from a binary search tree such as an AVL-tree or a red-black tree, nodes in a FT do not have a comparable key (and its associated value). Therefore, our tree-balancing algorithm only needs to hold one invariant—a balancing operation is triggered only if the height difference between any node’s left and right subtree is larger than 1. The FT implements four methods to handle all imbalance situations—*left_rotate*, *right_rotate*, *left_right_rotate*, and *right_left_rotate*. Due to the space limit, we omit the details of the tree balancing algorithms. Figure 6 and Figure 7 show the process of *right_rotate* and *right_left_rotate* operations, respectively.

3.3 Function Tree Integration

In this section, we describe how we integrate the FT scheme into Alibaba Cloud’s FaaS platform. The integration spans two components of our existing FaaS platform, the scheduler and the VM agent. Specifically, we integrate FAASNET’s FT manager into Alibaba Cloud’s FaaS scheduler and FAASNET’s VM worker into Alibaba Cloud’s FaaS VM agent, respectively (Figure 4). The scheduler manages VMs of a FT via the FT manager. The scheduler starts a FAASNET worker on each VM agent. A FAASNET worker is responsible for (1) serving scheduler’s commands to perform tasks of image

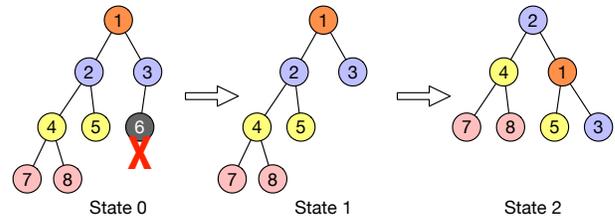


Figure 6: An example *right_rotate* operation. The FT manager detects that Node 6 was reclaimed and calls *delete* to remove it. Removal of Node 6 causes an imbalance, which triggers a *right_rotate* rebalancing operation. The FT manager then performs right rotation by marking Node 2 as the new root and marking Node 5 as Node 1’s left subtree.

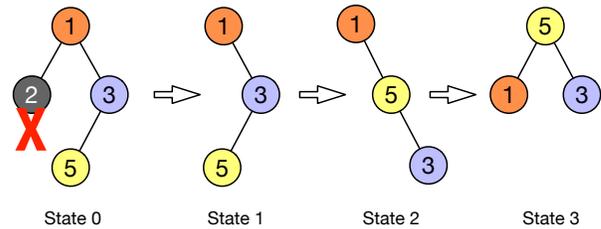


Figure 7: An example *right_left_rotate* operation. FT manager detects that Node 2 gets reclaimed and calls *delete* to remove it. Removal of Node 2 triggers a rebalancing operation. FT manager first right-rotates the right subtree of Node 1 by marking Node 5 as the parent of Node 3. FT manager then performs a *left_rotate* by marking Node 5 as the root.

downloading and container provisioning, and (2) managing the VM’s function containers.

FT Metadata Management. The scheduler maintains an in-memory mapping table that records the $\langle functionID, FT \rangle$ key-value pairs, which map a function ID to its associated FT data structure. A FT data structure manages a set of in-memory objects representing functions and VMs to keep track of information such as a VM’s *address:port*. The scheduler is sharded and is highly available. Each scheduler shard periodically synchronizes its in-memory metadata state with a distributed metadata server that runs *etcd* [11].

Function Placement on VMs. For efficiency, FAASNET allows one VM to hold multiple functions that belong to the same user. Function Compute uses a binpacking heuristic that assigns as many functions as possible in one VM host as long as the VM has enough memory to host the functions. As such, a VM may be involved in the topologies of multiple overlapping FTs. Figure 8 shows an example of a possible FT placement. In order to avoid network bottlenecks, FAASNET limit the number of functions that can be placed on a VM—in our deployment we set this limit to 20. We discuss a proposal of the FT-aware placement in §5.

Container Provisioning Protocol. We design a protocol to coordinate the RPC communications between the scheduler and FAASNET VM workers and facilitate container provisioning (Figure 9). On an invocation request, if the scheduler detects that there are not enough active VMs to serve the request

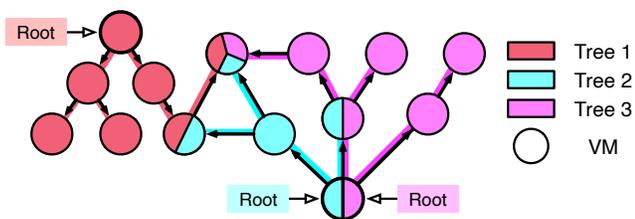


Figure 8: Example function placement on VMs. The color codings of trees and tree edges are red for tree 1 (left), blue for tree 2 (center), and purple for tree 3 (right). Arrows denote provisioning flows.

or all of current VMs are busy serving requests, the scheduler reserves one or multiple new VMs from the free VM pool and then enter the container provisioning process. Without loss of generality, we assume only one VM (VM_1) is reserved in this case. In Step 1, the scheduler creates a new metadata object for VM_1 and inserts it to the FT associated with the requested *functionID*. The scheduler then queries the FT in order to get the *address:port* of the upstream peer VM (VM_2). In Step 2, the scheduler sends the function metadata and *address:port* of VM_2 to VM_1 . Once receiving the information, VM_1 performs two tasks: (1) downloads the *.tar* manifest file of the function container image from the metadata store (§3.1), and (2) loads and inspects the manifest, fetches the URLs of the image layers, and persists the URL information on VM_1 's local storage. In Step 3, VM_1 replies back to the scheduler that it is ready to start creating the container runtime for the requested function. The scheduler receives the reply from VM_1 and then sends a *create container* RPC request to VM_1 in Step 4. In Step 5 and 6, VM_1 fetches the layers from upstream VM_2 based on the manifest configuration processed in Step 2. In Step 7, VM_1 sends the scheduler an RPC that the container has been created successfully.

FT Fault Tolerance. The scheduler pings VMs periodically and can quickly detect VM failures. If a VM is down, the scheduler notifies the FT manager to perform tree balancing operations in order to fix the FT topology.

3.4 FT Design Discussion

FAASNET offloads the metadata-heavy management tasks to the existing FaaS scheduler, so that each individual node in a FT serves the same role of fetching data from its parent peer (and seeding data for its child nodes if any). FT's root node does not have a parent peer but instead fetches data from the registry. FAASNET's FT design can completely eliminate the I/O traffic to the registry, as long as a FT has at least one active VM that stores the requested container. Earlier, our workload analysis reveals that a typical FaaS application would always have a throughput above 0 RPS (§2.2). This implies that, in practice, it is more likely for a request burst to scale out a FT from 1 to N rather than from 0 to N .

An alternative design is to manage the topology at finer-grained layer (i.e., blobs) granularity. In this approach, each individual layer forms a logical layer tree; layers that belong

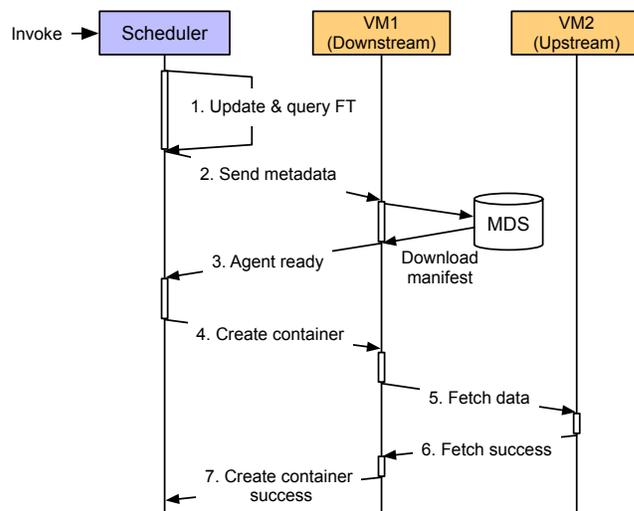


Figure 9: Container provisioning protocol. MDS: metadata store.

to a function container image may end up residing on different VMs. Note that FAASNET's FT is a special case of a layer tree model. Figure 10 shows an example. In this example, one VM stores layer files that belong to different function container images. Thus, a network bottleneck may occur when many downstream VM peers are concurrently fetching layers from this VM. This is because many overlapping layer trees form a fully-connected, all-to-all network topology. An all-to-all topology might scale well if VMs are connected with high-bandwidth network. However, the all-to-all topology can easily create network bottlenecks if each VM is resource-constrained, which is our case in Alibaba Cloud. We use small VMs with 2-core CPU, 4 GB memory, and 1 Gbps network in our FaaS infrastructure.

Existing container distribution techniques [16, 42] rely on powerful root node to serve a series of tasks including data seeding, metadata management, and P2P topology management. Porting these frameworks to our FaaS platform would require extra, dedicated, possibly sharded, root nodes, which would add unnecessary cost to the provider. FAASNET's FT design, on the other hand, keeps each VM worker's logic simple while offloading all logistics functions to our existing scheduler. This design naturally eliminates both the network I/O bottleneck and the root node bottleneck. In §4.3 and §4.4 we evaluate and compare FAASNET's FT design against a Kraken-like approach [16, 21], which adopts a layer-based topology with powerful root nodes.

3.5 Optimizations

We present the low-level optimizations that FAASNET uses to improve the efficiency of function container provisioning.

I/O Efficient Data Format. Regular `docker pull` and `docker start` are inefficient and time-consuming as the whole container image and the data of all the layers must be downloaded from a remote container registry [37] before

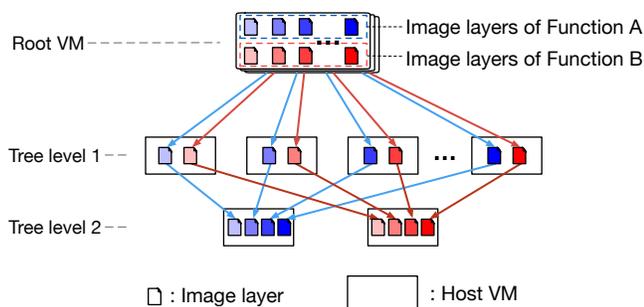


Figure 10: An example tree that manages the topology at layer granularity and relies on root node for data seeding and tree management.

the container can be started. To solve the issue, we design a new block-based image fetching mechanism within Alibaba Cloud. This mechanism uses an I/O efficient compression data file format. Original data is split into fixed-sized blocks and compressed separately. An offset table is used to record the offset of each compressed block in the compressed file.

FAASNET uses the same data format for managing and provisioning code packages. A code package is compressed into a binary file, which will be extracted by VM agent and eventually mounted inside of a function container. FAASNET distributes code packages the same way as it does for container images. §4.5 evaluates the performance benefit of I/O efficient data format on code package provisioning.

On-Demand I/O. For applications that do not need to read all the layers at once on startup, our block-based image fetching mechanism provides them with an option to fetch layer data at fine-grained block level, in a lazy manner (i.e., on-demand), from a remote storage (in our case, a container registry or a peer VM). First, the application, in our case, a FAASNET VM worker, downloads the image manifest file from a metadata store and does an image load locally to load the `.tar` image manifest. Second, it calculates the indices of the first and last (compressed) block and then consults with the offset table to find the offset information. Finally, it reads the compressed blocks and decompresses them until the amount of data that has been read matches the requested length. Since a read to the underlying (remote) block storage device must be aligned to the block boundary, the application may read and decompress more data than requested, causing read amplification. However, in practice, decompression algorithm achieves much higher data throughput than that of a block storage or network. Thus, trading extra CPU overhead for reduced I/O cost is beneficial in our usage scenario. We evaluate the effectiveness of on-demand I/O in §4.6.

RPC and Data Streaming. We build a user-level, zero-copy RPC library. This approach leverages non-blocking TCP `sendmsg` and `recvmsg` for transferring an `struct iovec` incontinuous buffer. The RPC library adds an RPC header directly to the buffer to achieve efficient, zero-copy serialization in the user space. The RPC library tags requests in order to achieve request pipelining and out-of-order receiving, similar

to HTTP/2’s multiplexing [14]. When a FAASNET worker receives a data block in its entirety, the worker immediately transfers the block to the downstream peer.

4 Evaluation

In this section, we evaluate FAASNET using production traces from Alibaba Cloud’s FaaS platform. We also validate FAASNET’s scalability and efficiency via microbenchmarks.

4.1 Experimental Methodology

We deploy FAASNET in Alibaba Cloud’s Function Compute platform using a medium-scale, 500-VM pool and a large-scale, 1,000-VM pool. We follow the same deployment configurations used by our production FaaS platform: all VMs use an instance type with 2 CPUs, 4 GB memory, 1 Gbps network; we maintain a free VM pool where FAASNET can reserve VM instances to launch cloud functions. This way, the container provisioning latency does not include the time to cold start a VM instance. FAASNET uses a block size of 512 KB for on-demand fetching and streaming. Unless otherwise specified: we use a function that runs a Python 3.8 PyStan application for about 2 seconds; the size of the function container image is 758 MB; the function is configured with 3008 MB memory; each VM runs one containerized function.

System Comparison. In our evaluation, we compare FAASNET against the following three configurations:

1. **Kraken:** Uber’s P2P-based registry system [16, 21]. We deploy a *Kraken* `devcluster` [17] with one origin node on our resource-constrained VM infrastructure.
2. **baseline:** Alibaba Cloud Function Compute’s current production setup. *baseline* downloads container images using vanilla `docker pull` from a centralized container registry.
3. **on-demand:** An optimized system based on *baseline* but fetches container layer data on demand (§3.5) from the container registry.
4. **DADI+P2P:** Alibaba’s DADI [1, 42] with P2P enabled. This approach uses one resource-constrained VM as the root node to manage the P2P topology.

Goals. We aim to answer the following questions:

1. Can FAASNET rapidly provision function containers under bursty FaaS workloads with minimum impact on workload performance (§4.2)?
2. Does FAASNET scale with increasing invocation concurrency levels (§4.3)?
3. How does function placement impact FAASNET’s efficiency (§4.4)?
4. How does FAASNET’s I/O efficient data format perform (§4.5)?
5. How effective is FAASNET’s on-demand fetching (§4.6)?

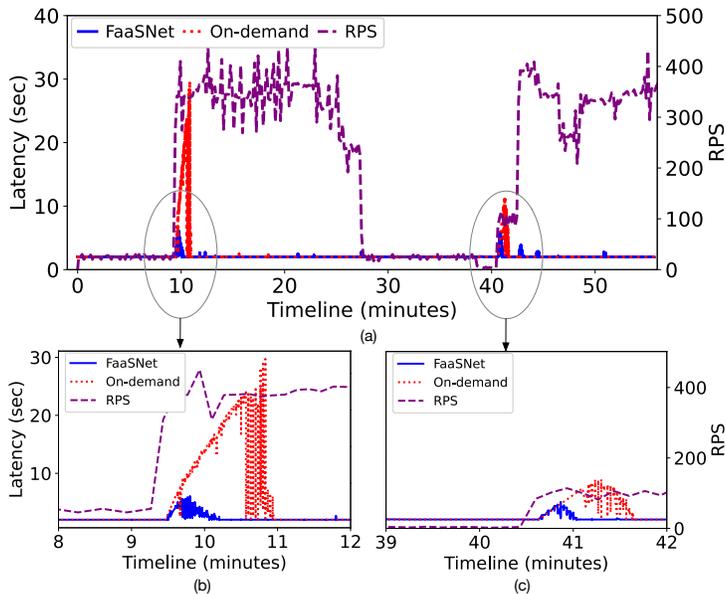


Figure 11: IoT trace timeline.

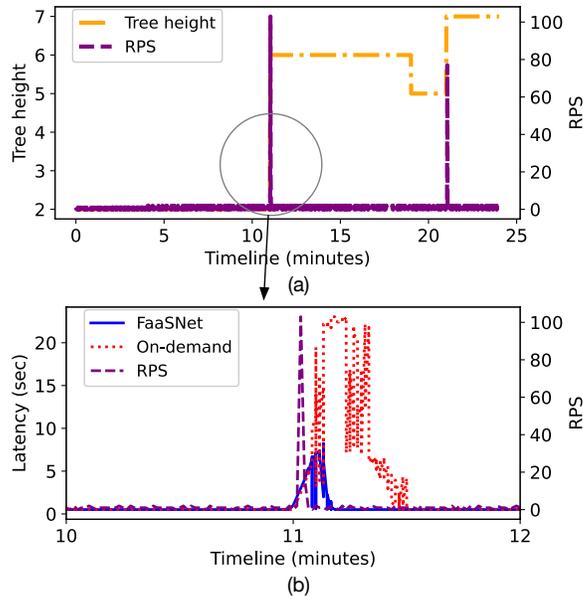


Figure 12: Synthetic trace timeline.

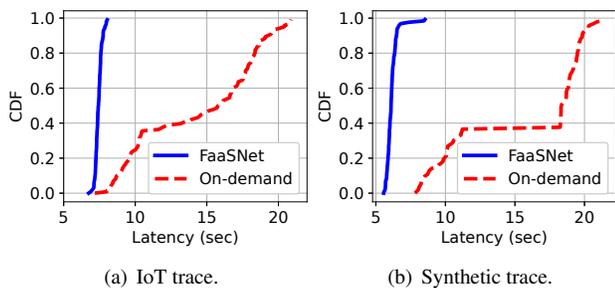


Figure 13: Distribution of container provisioning latency.

4.2 FaaS Application Workloads

In this section, we evaluate FAASNET using (scaled-down) application traces collected from our production workload (detailed in §2.2).

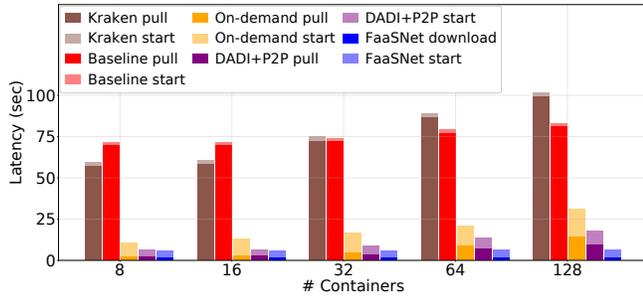
Trace Processing and Setup. We evaluate FAASNET using two FaaS applications: an IoT app and a gaming app. Since the original gaming workload exhibits a gradual ramp-up in throughput (Figure 2(a)), we instead create a synthetic bursty workload based on the gaming workload to simulate a sharp burst pattern for stress testing purpose. Our testing cluster has up to 1,000 VMs, so we scale down the peak throughput of both workload traces proportional (about 1/3 of the original throughput) to our cluster size and shorten the duration from 2 hours to less than 1 hour.

IoT Trace. The IoT trace exhibits two invocation request bursts. The first burst happens at 9 minute and the throughput increases from 10 RPS to 300-400 RPS; the peak throughput lasts for about 18 minutes and returns back to 10 RPS at 28 minute. The second burst happens at 40 minute and the throughput increases to 100 RPS, and then in about 2 minutes, jumps to around 400 RPS. Figure 11(a) plots the 55-minute timeline of the workload’s throughput and latency changes.

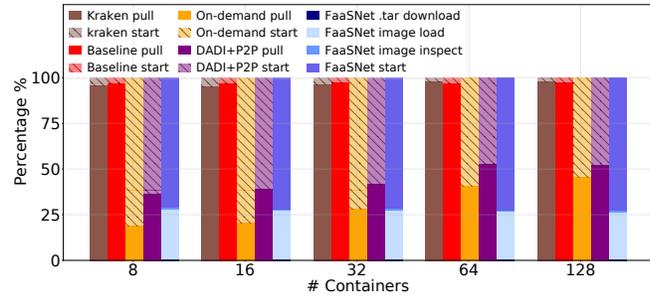
At 10 minute, the instantaneous throughput increase causes a backlog of function invocation requests at the FaaS scheduler side. Thus, the scheduler scales out the active VM pool by reserving a large number of free VMs from the free VM pool and starts the function container provisioning process. In *baseline* case, all newly reserved VMs start pulling container images from the registry, which creates a performance bottleneck at the registry side. As a result, the application-perceived response time—the end-to-end runtime that includes the container startup latency and the function execution time of around 2 seconds—increases from 2 seconds to about 28 seconds. Worse, the registry bottleneck inevitably prolongs the time that *baseline* requires to bring the response time back to normal. As shown in Figure 11(b), *baseline* finishes the whole container provisioning process and brings the response time back to normal in almost 113 seconds.

In contrast, FAASNET avoids the registry bottleneck—instead of downloading the container image from the registry, each newly reserved VM fetches image data block-by-block from its upstream peer in the FT, forming a data streaming pipeline. As long as a VM fetches enough data blocks, it starts the container. FAASNET reduces the maximum response time from 28 seconds to 6 seconds. Out of the 6 seconds, around 4 seconds are spent on fetching image layers from the upstream peer VM. (We present the container provisioning latency later in Figure 13.) More importantly, FAASNET requires only 28 seconds to bring the service back to normal, an improvement of 4× compared to the *on-demand* case.

Synthetic Trace. In the synthetic trace test, we simulate two function invocation request bursts and evaluate FT’s adaptivity. Figure 12(a) shows the timeline of a FAASNET FT’s height changes. At 11 minute, the throughput suddenly grows from 1 RPS to 100 RPS. FAASNET detects the burst and



(a) Average function container provisioning latency.



(b) Fraction of time spent at different stages.

Figure 14: Container provisioning scalability test.

rapidly scales the FT from a height of 2 (one root VM and one peer VM) to 7 (82 VMs in total). The FT starts parallel container provisioning instantly at 11 minute and sustains the latency spikes in about 10 seconds (Figure 12(b)). After the first burst, the throughput drops back to 1 RPS. Some VMs become cold and get reclaimed by the VM manager in about 15 minutes since the first burst. The number of VMs gradually reduces to 30 before the second burst arrives. Correspondingly, the height of the FT reduces from 6 to 5 (Figure 12(a)). When the second burst comes at 21 minute, the FT manager decides to grow the FT by adding another 62 VMs. With a total of 102 VMs, the height of the FT reaches up to 7 for serving the concurrent requests of the second burst.

Container Provisioning Cost We next analyze the container provisioning latency seen in the two workloads. As shown in Figure 13, since the registry in *on-demand* incurs a performance bottleneck, *on-demand* sees highly variant container provisioning latency, ranging from around 7 seconds to as high as 21 seconds. About 80% of the containers take at least 10 seconds to start. The container startup latency is highly predictable in FAASNET, with significantly less variation. For the synthetic workload, around 96% of the functions require only 5.8 seconds to start. For the IoT workload, almost all the functions start execution within a short time range between 6.8-7.9 seconds. This demonstrates that FAASNET can achieve predictable container startup latency.

4.3 Scalability and Efficiency

Next, we evaluate FAASNET’s scalability and efficiency via microbenchmarking.

Scaling Function Container Provisioning. In this test, we measure the time FAASNET takes to scale from 0 to N concurrent invocation requests, where N ranges from 8 to 128. Each invocation request creates a single container in a VM. Figure 14 reports the detailed results. As shown in Figure 14(a), *Kraken* performs slightly better than *baseline* under 8 and 16 concurrent requests but scales poorly under 32-128 concurrent requests. This is because *Kraken* distributes containers at layer granularity using a complex, all-to-all, P2P topology, which creates bottlenecks in the VMs. *Kraken* takes 100.4 seconds to launch 128 containers.

baseline achieves slightly better scalability than *Kraken*. The average container provisioning latency reaches up to 83.3 seconds when *baseline* concurrently starts 128 functions. Two factors contribute to the delay: (1) the registry becomes the bottleneck, and (2) *baseline*’s `docker pull` must pull the whole container image and layers (758 MB worth of data) from the registry and extract them locally.

Adding *on-demand* container provisioning to *baseline* improves the latency significantly. This is because *on-demand* eliminates most of the network I/Os for image layers that will not be instantly needed container startup. Despite pulling much less amounts of data from the registry, *on-demand* still suffers from the registry bottleneck; provisioning 128 function containers requires $2.9\times$ longer time than provisioning 8 containers in *on-demand* system.

DADI+P2P enables VMs to directly fetch image layers from peers, further avoiding downloading a large amount of layer blocks from the registry. However, *DADI+P2P* still has two bottlenecks: one at the registry side—image pulls are throttled at the registry, and layer-wise extract operation may also be delayed in a cascading manner by local VMs; the other bottleneck at the P2P root VM side—in addition to seeding data, the root VM in *DADI+P2P* is responsible for a series of extra tasks such as layer-tree topology establishment and coordination, thus forming a performance bottleneck. This can be evidenced from Figure 14(b) that the fractions of *DADI+P2P*’s image pull and container start maintain at the same level when scaling from 64 to 128 function starts.

Figure 14(a) shows that FAASNET scales perfectly well under high concurrency and achieves a speedup of $13.4\times$ than *baseline* and $16.3\times$ than *Kraken*. FAASNET is $5\times$ and $2.8\times$ faster than *on-demand* and *DADI+P2P*, respectively. As shown in Figure 14(b), FAASNET’s average container provisioning latency is dominated by two operations: image load and container start. FAASNET eliminates the bottleneck on both the two operations—on image load, FAASNET enables decentralized image loading (functionality-wise equivalent to image pull) at each VM by allowing each FAASNET worker to fetch the image manifest from the metadata store (with negligible overhead) and then starting the image loading process locally in parallel; on container start, each FAASNET VM

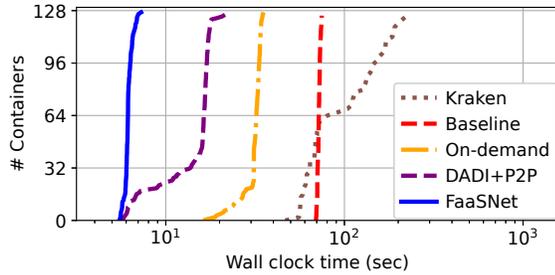


Figure 15: Container provisioning scalability test: wall clock time (X-axis) for starting N functions (Y-axis).

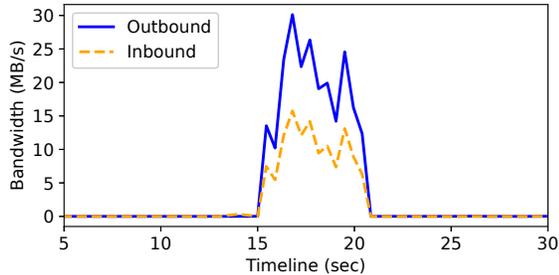


Figure 16: A timeline of the VM network bandwidth usage.

worker directly fetches layer blocks from peer VM and starts the function container once enough blocks are fetched. With all these optimizations, FAASNET maintains almost identical latency when scaling from 8 to 128 function startups.

Function Container Provisioning Pipeline. We next examine how long the whole container provisioning process spans. Figure 15 plots the timeline process that each system goes through to start N function containers. We only report the 128-function concurrency case. We observe that FAASNET starts the first function at 5.5 second and the 128th function at 7 second respectively. The whole container provisioning process spans a total of 1.5 seconds. Whereas *on-demand* and *DADI+P2P* span a total duration of 16.4 and 19 seconds, respectively. Specifically, it takes *DADI+P2P* a total of 22.3 seconds to start all the 128 containers, which is $14.7\times$ slower than that of FAASNET. This demonstrates that FAASNET’s FT-based container provisioning pipeline incurs minimum overhead and can efficiently bring up a large amount of function containers almost at the same time.

Figure 16 shows the bandwidth usage timeline for a VM that we randomly select from the 128-function concurrency test. Recall that a FAASNET worker along a FT path (i.e., not the root VM nor the leaf VM) performs two tasks: (1) fetches layer data from the upstream VM peer, and (2) seeds layer data to the two children VM peers in its downstream paths. We observe that the bandwidth usage of the inbound connection (fetching layers from upstream) is roughly half of that of the two outbound connections (sending layers to downstreams) during container provisioning. The aggregate peak network bandwidth is 45 MB/s, which is 35.2% of the maximum network bandwidth of the VM. We also observe that, the outbound network transfer is almost perfectly aligned

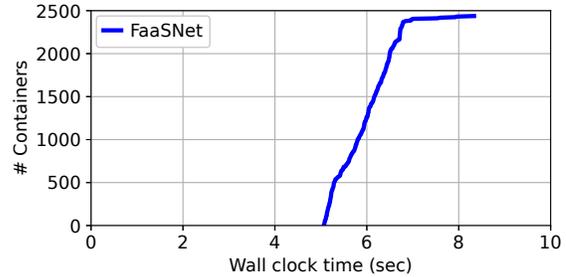


Figure 17: Large-scale function container provisioning: the wall clock time (X-axis) for starting N functions (Y-axis).

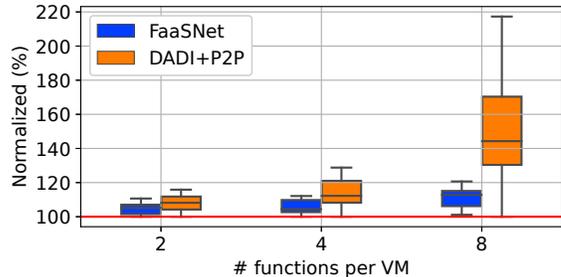


Figure 18: Container provisioning latency as a function of various function placement situations. For FAASNET and *DADI+P2P*, the latency is normalized to that of provisioning a single container in one VM in their own case.

with the inbound network transfer, again demonstrating the efficacy of FAASNET’s block-level data streaming scheme.

Large-Scale Function Startup. In this test, we create 1,000 VMs and concurrently invoke 2,500 functions on them. Each function uses a container of 428 MB and is configured to run with 1024 MB memory. Each VM runs two or three functions in this test. Figure 17 shows that all function containers finish provisioning and start running between 5.1 second and 8.3 second, again demonstrating FAASNET’s superb scalability. None of *on-demand* and *DADI+P2P* finishes the test due to timeout errors.

4.4 Impact of Function Placement

We conduct a sensitivity analysis to quantify the impact of function placement on container provisioning. In this test, we concurrently invoke 8 functions on N VMs, where N varies from 4 to 1. Each function has a different container (75.4 MB) and is configured to use 128 MB function memory (since a VM has 4 GB memory, it is allowed to host as much as 20 functions with 128 MB memory). We compare the container provisioning latency between FAASNET and *DADI+P2P*. As shown in Figure 18, *DADI+P2P* sees much higher latency variation when 4 functions and 8 functions are placed on the same VM, because *DADI+P2P*’s root VM is overloaded by the establishment processes of many small layer trees.

4.5 I/O Efficient Data Format

We next evaluate how the I/O efficient format helps with code package provisioning. We choose three functions: a simple, Python-written HelloWorld function that sleeps for 1 second

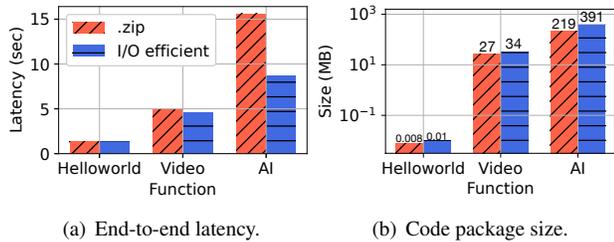


Figure 19: End-to-end invocation latency and code package size comparison between I/O efficient format and .zip.

(Helloworld), an FFmpeg video encoding function (Video), and a TensorFlow Serving function (AI), and compare FAASNET’s I/O efficient format with the baseline .zip format.

Figure 19(a) plots the end-to-end function invocation performance including the latency of code package downloading and function duration. Our I/O efficient format performs the same as .zip for Helloworld, since Helloworld’s code package has only 11 KB in size (Figure 19(b)). The I/O efficient format achieves better performance compared to .zip for Video and AI since the I/O efficient format fetches data on demand rather than extracting all data as .zip does. Figure 19(b) shows the code package sizes. Functions have a larger code package size when using I/O efficient format, because I/O efficient format’s compression incurs extra storage overhead.

4.6 On-Demand I/O: Sensitivity Analysis

Finally, we evaluate on-demand I/O and compare the impact of block sizes on read amplification. With on-demand fetching, a FAASNET worker only needs to fetch enough layer data blocks in order to start the function container. We choose three different function container images: (a) a 195 MB hello-world image with a Python 3.9 runtime pulled from Docker Hub; (b) a 428 MB PyStan image based on an AWS Lambda Python 3.8 base image; and (c) a 728 MB PyStan image based on an Alibaba Cloud Python 3.8 base image.

As shown in Figure 20, on-demand fetching can reduce the amount of data transferred via network. The reduction is especially profound for image b and c, because base images are dependency-heavy and are commonly used in the image building process. For example, with a block size of 512 KB (the block size configuration that we use in our evaluation), on-demand fetching sees a 83.9% reduction in network I/Os, compared to that of regular `docker pull`.

We also observe different levels of read amplification under different block sizes. This is because the starting and ending offset position is likely to be misaligned with the boundary of the (compressed) blocks in the underlying block device, the larger the block size is, the more useless data the FAASNET worker may read from the starting and ending blocks. The actual amount of data read (for starting a container) after decompression is even smaller, indicating that most of the dependencies included in the original container image is not used at the container startup phase. Exploring optimization



Figure 20: Amounts of data fetched (on-demand) as a function of block sizes. Left-most bar in each bar cluster represents the size of the original container image; right-most bar in each bar cluster represents the actual amount of data read from the data blocks fetched via network.

to reduce the read amplification is part of our future work.

5 Discussion

In this section, we discuss the limitations and possible future directions of FAASNET.

FT-aware Placement. When the number of functions grows, the contention of network bandwidth ensues. Though §4.4 proves it less a concern in FAASNET than prior work, for the sake of safety in production, we program the system to avoid co-locating multiple functions if the cluster resources permit. Anticipating a future demand increase of custom containers, we plan to address the problem by extending the container placement logic. The general goal is to balance the inbound and outbound communication of each VM when multiple functions are being provisioned. Intuitively, by adjusting container placement, we can control the number of FTs that a VM is involved in and the role (e.g., leaf vs. interior node) a VM serves, and thus the bandwidth consumption. A further optimization is to co-locate functions that share common layers, so they could reduce the amount of data transfer.

Multi-Tenancy. As mentioned, Alibaba Cloud achieves strong, tenant-level function isolation using containers and VMs. As such, our FaaS platform cannot share VMs among tenants. This means that FAASNET’s FTs are naturally isolated between different tenants. Porting FAASNET to other secure and lightweight virtualization techniques [24, 45, 52, 57] is our ongoing work.

FAASNET for Data Sharing. Technically, our work can enable the sharing of container images among VMs through P2P communication, There is potentiality for it to generalize to a broader scope: data sharing for general container orchestration systems such as Kubernetes [27]. Such a need is arising in FaaS platforms with the emergence of data-intensive applications, such as matrix computation [31, 51], data analytics [39, 49], video processing [26, 35], and machine learning [30, 38], etc. Most of them rely on a centralized storage for data exchange, which is a similar bottleneck as the container registry in our work. Hence we believe the design of FAASNET can also accelerate data sharing, only with two additional challenges: (1) how to design a primitive interface

for users; (2) how to adapt the tree management algorithms for more frequent topology building and change. We leave the exploration as a future work.

Adversarial Workloads. Extremely short-lived functions with a duration at sub-second level and sparse invocations may be adversarial to FAASNET and custom-container-based FaaS platforms. Function environment caching and pre-provisioning [22, 48, 50] can be used to handle such workloads but with extra infrastructure-level costs.

Portability. FAASNET is transparent to both upper-level FaaS applications and underlying FaaS infrastructure. It reuses Function Compute’s existing VM reclaiming policy and could be applied to other FaaS platforms without introducing extra system-level costs. Porting FAASNET to Alibaba Cloud’s bare-metal infrastructure is our ongoing work.

6 Related Work

Function Environment Caching and Pre-provisioning. FaaS applications face a notoriously persisting problem of high latency—the so-called “cold start” penalty—when function invocation requests must wait for the functions to start. Considerable prior work has examined ways to mitigate the cold start latency in FaaS platforms. FaaS providers such as AWS Lambda and Google Cloud Functions pause and cache invoked functions for a fixed period of time to reduce the number of cold starts [22, 55, 56]. This would, however, increase the TCO for providers. To reduce such cost, researchers propose prediction methods that pre-warm functions just in time so that incoming recurring requests would likely hit on warm containers [50]. SAND shares container runtimes for some or all of the functions of a workflow for improved data locality and reduced function startup cost [25]. SOCK caches Python containers with pre-imported packages and clones cached containers for minimizing function startup latency [48]. PCPM pre-provisions networking resources and dynamically binds them to function containers to reduce the function startup cost [46]. While function requests can be quickly served using pre-provisioned, or cached, virtualized environments, these solutions cannot fundamentally solve the issue of high costs incurred during function environment provisioning.

Sandbox, OS, and Language-level Support. A line of work proposes low-level optimizations to mitigate FaaS cold start penalty. Catalyzer [34] and SEUSS [29] reduce the function initialization overhead by booting function instances from sandbox images created from checkpoints or snapshots. Systems such as Faasm [53] and [28] leverage lightweight language-based isolation to achieve speedy function startups. Unlike FAASNET, these solutions either require modified OSes [29, 34] or have limited compatibility and usability in terms of programming languages [28, 53].

Container Storage. Researchers have looked at optimizing container image storage and retrieval. Slacker speeds up the container startup time by utilizing lazy cloning and

lazy propagation [37]. Images are stored and fetched from a shared network file system (NFS) and referenced from a container registry. Wharf [61] and CFS [44] store container image layers in distributed file systems. Bolt provides registry-level caching for performance improvement [43]. These work are orthogonal in that FAASNET can use them as backend container stores. Kraken [16] and DADI [42] use P2P to accelerate container layer distribution. These systems assume a static P2P topology and require dedicated components for image storage, layer seeding, or metadata management, which leave them vulnerable to high dynamicity (demanding high adaptability of the network topology) and unpredictable bursts (requiring highly scalable container distribution).

AWS Lambda Containers. AWS announced the launch of container image support for AWS Lambda [19] on December 01, 2020. Limited information was revealed via a re:Invent 2020 talk [6] about this feature: AWS uses multi-layer caching to aggressively cache image blocks: (1) microVM-local cache, (2) shared, bare-metal server cache, and (3) shared, availability zone cache. The solution, while working for powerful, bare-metal server-based clusters that can co-locate many microVMs [24], is not suitable for our FaaS platform, which is based on thousands of small VMs managed by Alibaba Cloud’s public cloud platform.

P2P Content Distribution. VMThunder uses a tree-structured P2P overlay for accelerating VM image distribution [59]. A BitTorrent-like P2P protocol is proposed for achieving similar goals [33]. Bullet uses an overlay mesh for high-bandwidth, cross-Internet file distribution [41]. FAASNET builds on these works but differs with a new design that is attuned to the FaaS workloads.

7 Conclusion

Scalable and fast container provisioning can enable fundamental elasticity for FaaS providers that support custom-container-based cloud functions. FAASNET is the first system that provides an end-to-end, integrated solution for FaaS-optimized container runtime provisioning. FAASNET uses lightweight, decentralized, and adaptive function trees to avoid major platform bottlenecks. FAASNET provides a concrete solution that is attuned to the requirements of a large cloud provider’s FaaS platform (Alibaba Cloud Function Compute). We show via experimental evaluation that FAASNET can start thousands of large function containers in seconds. Our hope is that this work will make container-based FaaS platforms truly elastic and open doors to a broader class of dependency-heavy FaaS applications including machine learning and big data analytics.

To facilitate future research and engineering efforts, we release the source code of FAASNET’s FT prototype as well as an anonymized dataset containing production FaaS cold start traces collected from Alibaba Cloud Function Compute at: <https://github.com/mason-leap-lab/FaaSNet>.

Acknowledgments

We are grateful to our shepherd, Jon Crowcroft, and the anonymous reviewers, for their valuable feedback and suggestions. We would like to thank Benjamin Carver for his proofreading, and Jianlie Zou, Qi Sun, and Yifan Yuan for their kind help on this work. This work is sponsored in part by NSF under an NSF CAREER Award CNS-2045680, CCF-1919075, and CCF-1919113; Ao Wang is supported by Alibaba Group through the Alibaba Research Intern Program.

References

- [1] Alibaba: Accelerated Container Image. <https://github.com/alibaba/accelerated-container-image>.
- [2] Alibaba Cloud Function Compute. <https://www.alibabacloud.com/product/function-compute>.
- [3] Alibaba Cloud Function Compute Custom Container Runtime. <https://www.alibabacloud.com/help/doc-detail/179368.htm>.
- [4] Alibaba Cloud Function Compute's custom container doc. <https://github.com/awesome-fc/custom-container-docs>.
- [5] AWS Lambda. <https://aws.amazon.com/lambda/>.
- [6] AWS re:Invent 2020: Deep dive into AWS Lambda security: Function isolation. <https://www.youtube.com/watch?v=FTwsMYXWGB0>.
- [7] Azure Functions. <https://azure.microsoft.com/en-us/services/functions/>.
- [8] Deploy Lambda functions with .zip file archives. <https://docs.aws.amazon.com/lambda/latest/dg/python-package.html>.
- [9] Docker. <https://www.docker.com/>.
- [10] Dragonfly: An Open-source P2P-based Image and File Distribution System. <https://d7y.io/en-us/>.
- [11] etcd: A distributed, reliable key-value store for the most critical data of a distributed system. <https://etcd.io/>.
- [12] Google Cloud Functions. <https://cloud.google.com/functions/>.
- [13] Google Cloud Run. <https://cloud.google.com/run>.
- [14] HTTP/2. <https://http2.github.io/>.
- [15] Knative: Kubernetes-based platform to deploy and manage modern serverless workloads. <https://knative.dev/>.
- [16] Kraken: A P2P-powered Docker registry that focuses on scalability and availability. <https://github.com/uber/kraken>.
- [17] Kraken devcluster deployment. <https://github.com/uber/kraken/tree/master/examples/devcluster>.
- [18] Kubernetes: Performing a Rolling Update. <https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/>.
- [19] New for AWS Lambda – Container Image Support. <https://aws.amazon.com/blogs/aws/new-for-aws-lambda-container-image-support/>.
- [20] OpenFaaS. <https://www.openfaas.com/>.
- [21] Uber Releases Kraken: An Open Source P2P Docker Registry. <https://www.infoq.com/news/2019/04/uber-kraken-p2p-docker/>.
- [22] Understanding Container Reuse in AWS Lambda. <https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/>.
- [23] 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.
- [24] 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.
- [25] 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, 2018. USENIX Association.
- [26] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 263–274, New York, NY, USA, 2018. ACM.
- [27] The Kubernetes Authors. Production-grade container orchestration - kubernetes. <https://kubernetes.io/>.
- [28] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 645–650, Boston, MA, July 2018. USENIX Association.

- [29] 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.
- [30] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC '19*, pages 13–24, New York, NY, USA, 2019. ACM.
- [31] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20*, page 1–15, New York, NY, USA, 2020. Association for Computing Machinery.
- [32] Benjamin Carver, Jingyuan Zhang, Ao Wang, and Yue Cheng. In search of a fast and efficient serverless dag engine. In *4th International Parallel Data Systems Workshop (PDSW 2019)*, 2019.
- [33] Zhijia Chen, Yang Zhao, Xin Miao, Ying Chen, and Qingbo Wang. Rapid provisioning of cloud infrastructure leveraging peer-to-peer networks. In *2009 29th IEEE International Conference on Distributed Computing Systems Workshops*, pages 324–329, 2009.
- [34] 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.
- [35] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, 2017. USENIX Association.
- [36] Jim Gray. Why do computers stop and what can be done about it?, 1985.
- [37] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 181–195, Santa Clara, CA, 2016. USENIX Association.
- [38] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 2021)*, June 2021.
- [39] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99th. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC '17*, pages 445–451, New York, NY, USA, 2017. ACM.
- [40] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A Berkeley view on serverless computing. Technical Report UCB/Eecs-2019-3, Eecs Department, University of California, Berkeley, Feb 2019.
- [41] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03*, page 282–297, New York, NY, USA, 2003. Association for Computing Machinery.
- [42] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. DADI: Block-level image service for agile and elastic application deployment. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 727–740. USENIX Association, July 2020.
- [43] Michael Littlely, Ali Anwar, Hannan Fayyaz, Zeshan Fayyaz, Vasily Tarasov, Lukas Rupperecht, Dimitrios Skourtis, Mohamed Mohamed, Heiko Ludwig, Yue Cheng, and Ali R. Butt. Bolt: Towards a scalable docker registry via hyperconvergence. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 358–366, 2019.
- [44] Haifeng Liu, Wei Ding, Yuan Chen, Weilong Guo, Shuoran Liu, Tianpeng Li, Mofei Zhang, Jianxing Zhao, Hongyin Zhu, and Zhengyi Zhu. Cfs: A distributed file system for large scale container platforms. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19*, page 1729–1742, New York, NY, USA, 2019. Association for Computing Machinery.
- [45] 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.
- [46] 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.
- [47] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.
- [48] 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, 2018. USENIX Association.
- [49] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, 2019. USENIX Association.
- [50] Mohammad Shahradd, 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.
- [51] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 281–295, New York, NY, USA, 2020. Association for Computing Machinery.
- [52] 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 '19, page 121–135, New York, NY, USA, 2019. Association for Computing Machinery.
- [53] 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.
- [54] Markus Thömmes. Squeezing the milliseconds: How to make serverless platforms blazing fast! <https://googl/zvqtBP>.
- [55] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupperecht, 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.
- [56] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, 2018. USENIX Association.
- [57] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 199–211, New York, NY, USA, 2018. Association for Computing Machinery.
- [58] 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 *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.
- [59] Zhaoning Zhang, Ziyang Li, Kui Wu, Dongsheng Li, Huiba Li, Yuxing Peng, and Xicheng Lu. Vmthunder: Fast provisioning of large-scale virtual machine clusters. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3328–3338, 2014.
- [60] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupperecht, Dimitrios Skourtis, Amit S Warke, Mohamed Mohamed, and Ali R Butt. Large-scale analysis of the docker hub dataset. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10. IEEE, 2019.
- [61] Chao Zheng, Lukas Rupperecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit S. Warke, and Dean Hildebrand. Wharf: Sharing docker images in a distributed file system. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 174–185, New York, NY, USA, 2018. Association for Computing Machinery.

Experiences in Managing the Performance and Reliability of a Large-Scale Genomics Cloud Platform

Michael Hao Tong, Robert L. Grossman, Haryadi S. Gunawi
University of Chicago

Abstract

We share our technical experiences in improving the performance of long-running jobs on the Genomic Data Commons (GDC), a large-scale cancer genomics cloud platform. We show how common bioinformatics workloads can cause VMs to age after several days, causing a large number of Extended Page Table (EPT) violations that significantly impact performance. We present host- and VM-level EPT monitoring and evaluate several possible mitigation scenarios. We highlight the long investigative process required for this research, with experiments requiring many days to complete.

1 Introduction

Since the invention of DNA sequencing in 1977, tremendous volumes of genomic sequencing data have been produced, and the number of biological databases has been doubling about every 15 months [1]. A driving force behind this is the exponential drop in the cost for sequencing a human genome, which is shown by the solid purple line in Figure 1 [2]. As can be seen from Figure 1, the cost of sequencing a genome has been decreasing faster than Moore’s Law (shown by the dashed green line).

With this tremendous growth in biological data, it has become more challenging to manage and analyze the data. As a result, in the recent years, a wide range of bioinformatics methods, techniques, algorithms and tools have been developed to analyze the experimental data and understand the underlying biological mechanisms and their significance [3].

Genomic sequencing data is particularly important in cancer, since cancer is in large part driven by genomic mutations. The GDC was launched in 2016 with the goal of providing a repository for cancer genomics data and for harmonizing data that is submitted to it with a common set of bioinformatics pipelines. The GDC is large-scale cloud based system that stores, analyzes, and shares genomic, clinical and imaging data from patients with cancer. The GDC is a hybrid cloud system and uses both a private on-premise cloud and the public AWS cloud. The GDC is one of the

largest bioinformatics platforms that support the cancer research community (see Section 2 for more details).

An important backbone of the GDC is GPAS, the GDC Pipeline Automation System that is used for processing data submitted to the GDC and running a wide range of bioinformatics pipelines over the data. It is important to note that at the scale the GPAS operates, using public compute and storage clouds would be 1.5x or more expensive than using private on-premise clouds. On the other hand, using public clouds is important for GPAS for burst computing, for making use of a wider variety of machine configurations, for running portions of larger, more complex pipelines, and for flexibility in general.

On the compute side, GPAS runs a large on-premise VM cloud powered by OpenStack/KVM, and the on-premise GPAS clusters uses a combination of Ceph and Cleversafe for storage. While there are many interesting experiences to share, this paper focuses on technical matters that might benefit the systems community. In particular, we present our experiences in managing the bioinformatics pipelines and KVM performance in GPAS.

KVM performance (EPT violations under extreme memory fragmentation). We reveal a significant VM performance problem that surfaced in GPAS. We noticed that a significant number of jobs in GPAS exhibited much slower performance compared to other similar jobs, with a *degradation of up to 10x*.

GPAS workloads are I/O intensive, require large memory instances, and, most importantly, are *long running*. For example, many jobs take weeks to complete. These characteristics made the problem difficult to troubleshoot. Online monitoring tools that we typically use only report high-level aggregate metrics and did not pinpoint the root cause. Trying to replicate the problem in an “offline” setting also did not reveal the root cause because of the different environments (fresh vs. aged VMs). Recording more online metrics did not give us quick outcomes because the problem did not appear in the early days of the jobs.

Because of all of these observations, we speculated that the problem could be related to memory fragmentation of aging VMs where many sequential guest pages are not mapped sequentially on the physical pages, causing extreme translation lookaside buffer (TLB) misses. More specifically, this brought us to the root cause, the overhead of hardware-

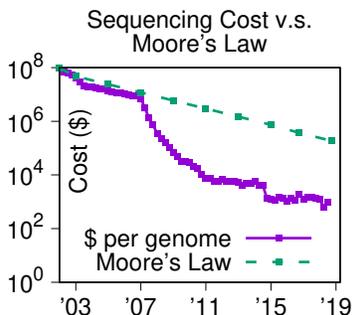


Figure 1: Sequencing cost.

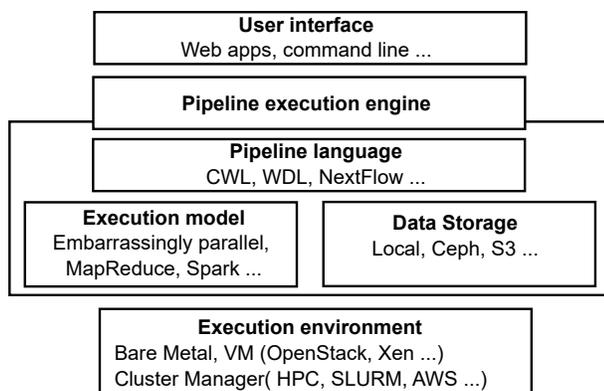


Figure 2: **Full stack bioinformatics pipeline platform.**

assisted Intel Extended Page Table (EPT) [4] technology used by the virtual machine memory management in the VM hypervisor [5]. Interestingly, we confirmed that the problem only surfaces in the OpenStack/KVM stack. We were not able to reproduce the problem in cloud VMs used by Amazon Web Services (AWS) and Google Cloud Platform (GCP). We speculate that public clouds might use their own proprietary virtualization technology (that is not available to us to analyze).

Besides presenting our findings, this paper also shows challenges in monitoring EPT performance issues. We also present several mitigation scenarios that we tried out, such as rebooting the VMs, defragmenting the memory, running on bare metal, and using public clouds, along with their advantages and disadvantages.

2 Background and Motivation

Figure 2 illustrates the full stack of layers of a typical bioinformatics cloud platform from the user/administrative interface to the execution platform. In order to improve portability and replicability, bioinformatics pipelines are often written in a workflow language [6], such as the Common Workflow Language (CWL) [7] or the Workflow Description Language (WDL) [8]. Bioinformatics pipelines are also typically containerized. There are several systems available for executing workflows expressed in workflow languages, including CWLtool [9] for CWL and Cromwell for WDL [8]. As this paper discusses the core systems aspect of GPAS, we have put additional information about the pipelines in the supplemental material [10]. To improve isolation to support security requirements, GPAS runs each containerized pipeline in a virtual machine. GPAS manages virtual machines using OpenStack/KVM. GPAS currently uses CWLtool. CWLtool does not support parallelizing tasks of a pipeline across different machines, however tasks can run in parallel across cores of a machine. Currently, GPAS uses a policy of allocating one VM per physical node, since many GDC pipelines benefit from this allocation. GPAS uses SLURM as the cluster manager for the VM pool and sched-

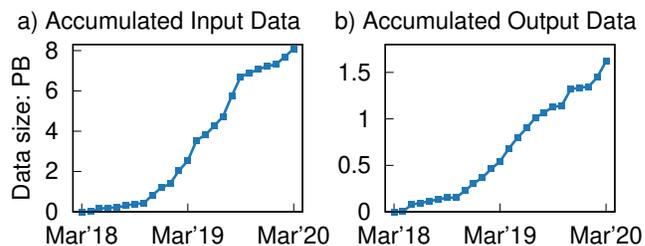


Figure 3: **Accumulated input consumption and output production over the months, §2.**

Clouds	Approx. Hourly	Approx. Annual
On premise	\$0.31	\$691,000
AWS (Spot)	\$0.75	\$1,693,000
AWS (3-yr Reserved)	\$1.06	\$2,386,000
AWS (On-Demand)	\$2.50	\$5,645,000

Table 1: **The approximate costs of on-premise vs. public cloud, §2.** For AWS, we assume that EC2 *i3.xlarge* instances are used, which are roughly similar to the on-premise instances.

ules jobs to run on SLURM. In bioinformatics, the terms “pipeline” and “workflow” are often used interchangeably.

On average, over 3,000 researchers use GDC every day, with over 100,000 unique researchers using the GDC each year. Over 1 PB of data is accessed or downloaded from the GDC in a typical month. GDC currently manages over 15 PB of data across almost 100 storage servers, including the released data, the data being processed for release, temporary files, and backup copies. There are over 200 compute nodes containing 7500 cores, 50 TB of RAM, and 900 TB of local disk/SSD storage. Figures 3a and 3b show the accumulated amount of data that GPAS has processed and produced by month for a two year period. Interested readers can refer to the GDC Documentation [11] for more details.

The GPAS on-premise cloud annually processes about 322,000 workflows requiring 35 million core hours. At this scale, the costs using a public cloud would be significantly higher, as shown in Table 1. For this comparison, on-premise costs assume a 4-year amortization of equipment costs and a 15-year amortization of data center costs. The comparison assumes that on-premise nodes are used at 100% utilization.

3 Workloads in GPAS

3.1 The bioinformatics pipelines and tools

GPAS uses a wide range of pipelines that include various bioinformatics analytical tools and serve different analysis purposes. At this time, there are 10 open-sourced pipelines in the GDC GitHub repository [12].

Table 2 lists the open-source pipelines and the main tools used in these pipeline. GPAS uses a large variety of tools that include in-house software or scripts, such as HTSeq Tool and GDC VEP Tool, and widely used third-party bioinformatics tools, such as BWA and GATK. In addition to those listed in

Pipeline	Main tools used
DNA-Seq Alignment	BWA [13], Biobambam2 [14], Picard Tools [15], GATK [16]
RNA-Seq Alignment	STAR [17]
miRNA Alignment and Profiling	BWA [13]
RNA-Seq HTSeq Quantification	HTSeq Tool [18]
WXS Variant Calling	MuSE [19], SomaticSniper [20], VarScan2 [21], MuTect2 [22]
WXS Variant Filtering	Picard Tools [15]
WGS Variant Calling	CGP WGS [23]
VEP Variant Annotaton	GDC VEP Tool [24]
Mutation Annotation File (MAF) Generation	MAF Tools [25]
SNP6 Segmentation	snp6cbs [26]

Table 2: **Open-source GDC Pipelines and the main tools used in the pipelines.**

Tool	Language	I/O Intensity	Computing Intensity
samtools	C	High	Low
BWA	C	High	High
FastQC	Java	Medium	Low
MuSE	C++	Low	High
MuTect2	Java	Low	High
SomaticSniper	C	High	High
VarScan2	Java	High	High

Table 3: **Characteristics of the tools used in GPAS.** *The observations are based on the way we used the tools in GPAS, the results may differ if they are used in different ways*

the table, FastQC [27] and samtools [28] are also commonly used across all the GDC pipelines.

Due to the large number of tools that exist in bioinformatics, it is difficult to characterize the computing resource demand and performance of each tool before putting them into use. We list the the characteristics of some tools in Table 3 that are used in the DNA-Seq alignment and whole exome sequencing (WXS) variant calling workflows, two of the longer running workflows in GPAS. Note that the way GPAS uses the tools in Table 3 is that, for samtools, FastQC, MuSE, MuTect2, SomaticSniper, VarScan2, GPAS spawns multiple processes each of which runs the same tool on separate input files, even though the tool may support multi-threading by itself [10]; for BWA, GPAS uses BWA’s own multi-threading functionality.

Table 3 shows that even though MuSE, MuTect2, SomaticSniper and VarScan2 are all somatic variant calling tools, due to different algorithms and program designs, they expose different patterns in I/O intensity. The GPAS pipelines are quite varied, and some GPAS pipelines, such as the MAF generation pipeline, require significantly less time to complete.

3.2 Pipeline Job Performance

For simplicity of discussion, we mainly measure job-level performance. A pipeline job is an execution of the pipeline that handles a specific set of input data. Readers who are interested in knowing the logical abstraction and composition of the pipeline can read our extended report [10].

The majority of the jobs in GPAS process a large amount of input data and take a long time to complete. Thus, a simple metric for understanding job performance and its degradation is the processing rate:

$$processing\ rate = \frac{job\ execution\ time}{input\ data\ size} \quad (1)$$

The unit of *processing rate* is *seconds/GB* or *hours/GB*, thus a larger *processing rate* value means means the performance is worse (as more time is needed for processing one GB of input data).

Due to the complexity of bioinformatics pipelines in GPAS, the job performance shows quite interesting patterns. The job performance shown in this section are collected from jobs that ran on bare metal nodes, so that we avoid the interference from VMs (which will be discussed later).

Through linear regression and significance-test analysis, it is found that, among all the parameters known at the time a job starts (such as number of input files, number of CPUs allocated, etc.), input data size exhibits the strongest correlation with *processing rate*.

To better illustrate the correlation between input size and *processing rate*, jobs are sorted by their *processing rate*, and divided into 20 buckets according to the percentile of *processing rate*. Average input size of each bucket is shown in Figure 4 and 5. There can be either positive or negative correlation between input size and *processing rate*, depending upon the specific pipeline. There are three relationships that emerge from this analysis:

Larger input size means slower/larger processing rate (positive correlation): As shown in Figure 4, *processing rate* is positively correlated with input size for some pipelines. This is the case for sequence alignment pipelines. Note that, as can be seen in the graph to the right, *processing rate* of this pipeline is generally very large, more than 1500 seconds/GB. Sequence alignment involves multiple string searches and exhibits positive correlation between input data size and *processing rate*.

Larger input size means faster/smaller processing rate (negative correlation): Contrary to the previous result, Figures 5 shows that *processing rate* is negatively correlated with input size for some pipelines. For example, this is the case for some utility pipelines that involve accessing and extracting files from cloud buckets for processing. Note that *processing rate* in this pipeline is much faster than for the jobs in the previous one. With a high-bandwidth network and I/O, *processing rate* is higher. However, since the execution time for such pipeline jobs is short, the overhead of

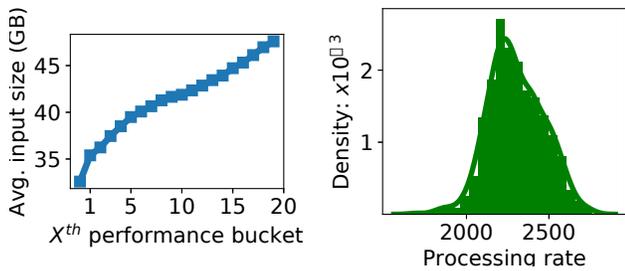


Figure 4: **A DNA-Seq alignment pipeline shows larger input size has slower/larger processing rate.** Each performance bucket contains 5% jobs, and buckets that have larger number demonstrate worse performance, §3.2

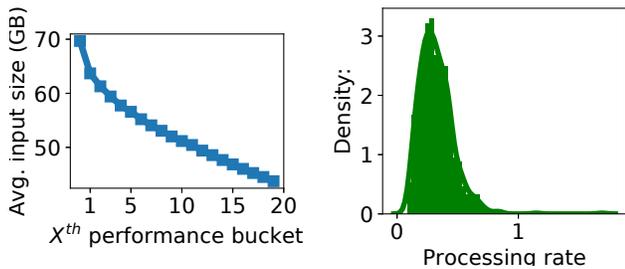


Figure 5: **An internal light-weighted pipeline shows larger input size has faster/smaller processing rate.** Each performance bucket contains 5% jobs, and buckets that have larger number demonstrate worse performance, §3.2

spawning containers and other components is significant in comparison. As a result, smaller input size leads to worse processing rate.

External factors interference: In addition, there are some cases that may not comply with the previous behaviors. Figure 6 shows a distribution with two peaks for the job processing rate of a pipeline. By inspecting job logs, it is found that jobs at the right peak (slower jobs) spent significantly longer times downloading data. That suggests that there might have been network congestion in the cluster during that time. Hence, in some cases, we need to take the external environment into account to understand the processing rate of jobs.

To conclude, through statistical analysis of the jobs, although there are some degrees of variation, we find that processing rate for pipelines is highly correlated with a job's input size. The exact correlation depends on the workloads associated with the pipeline. Sometimes, the processing rate might also be subject to external factors during some parts in the job. It is important to understand the nature of the job so that good interpretation can be formed.

In the next section, we will discuss performance issues that arise and create long tail distributions when jobs are executed on VMs.

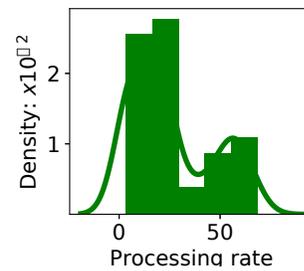


Figure 6: **Abnormal Processing rate Distribution.** Each performance bucket contains 5% jobs, and buckets that have larger number demonstrate worse performance, §3.2

4 Performance Issues: Aging VMs and EPT Violations

The GDC uses over 20 complex pipelines for processing data. Some of the pipeline components are I/O bound and some are memory bound. Complicating matters, some of the pipelines are long-running and can take several weeks to complete. After several months, it became apparent that some of the pipelines failed to complete, especially the longer running ones, which required retrying the pipelines, and which decreased the overall throughput of the system. After some experimentation, it appeared that long running pipelines performed better on bare metal nodes versus using virtual machines, and some of the nodes running virtual machines were replaced with bare metal nodes for this reason.

Figure 7a shows the success rate for jobs in each month. GPAS serves more than 20,000 jobs per month and has achieved over 90% monthly success rate for the jobs. The figure shows the success rate of jobs running on VMs (red) and bare metal (green). As noted, for long running jobs (those that require weeks), performance, as measured by GB processed per hour, tends to decline, and failures tend to become more common. Some jobs have to be stopped in the middle due to their very poor performance.

Starting in June 2019, GPAS began to run some jobs on bare metal nodes, while continuing to run most jobs on virtual machines. Although using bare metal nodes improves job completion rates, it significantly complicates the management of GPAS, since the GDC in general is designed around the setup, management, and monitoring of virtual nodes. For this reason, identifying the root cause of the higher failure rate for long-running jobs and mitigation strategies for improving the completion rate was important.

4.1 Job Performance Variance

Our next step was to quantify the slowdown. Originally, the GPAS compute pool used only VMs managed by OpenStack.

We divide pipeline jobs into comparable groups [10], and Figure 7b shows the cumulative distribution function (CDF) of processing rate of jobs on VMs and bare metal nodes that

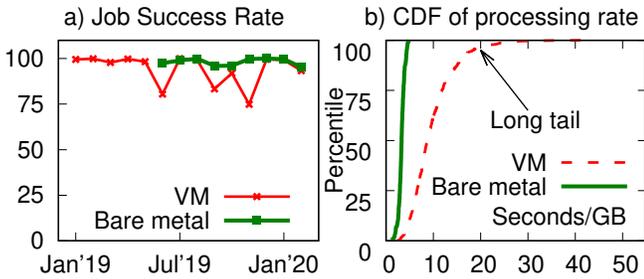


Figure 7: **Performance of jobs running on VMs and bare metal nodes, §4.** *a)* shows the success rate over the months for jobs running on VMs and bare metal nodes (which were added to the computing pool at a later time); *b)* shows processing rate CDF comparison between VMs and bare metal nodes.

we profiled in our deployment for one of the groups. For this figure, there are 796 and 247 jobs on VMs and bare metal, respectively, and all the jobs run the same type of pipeline (called “variant-filtration.pindel”), hence all the jobs should observe similar performance. In Figure 7a, the nearly vertical solid line represents *stable processing rate* of jobs on bare metal nodes. However, the dashed line of Figure 7b shows that *processing rate* on VMs is generally worse than *processing rate* on bare metal nodes. While this is expected, the issue lies in the *tail* performance especially at high percentiles. According to the zoomed graph in Figure 7b, starting from the 80th percentile, *processing rate* on VMs is 3 times worse than bare metal nodes, and in higher percentiles, the performance gets worse by an even larger magnitude.

We also would like to note again that the jobs in Figure 7b are jobs that come from one type of job pipeline (“variant-filtration.pindel”) that generally only spends not more than 40 seconds/GB. However, there are other more CPU/memory-intensive pipelines that spend 1000 – 3000 seconds for each GB. The problem raised in this paper becomes worse for these even more intensive pipelines.

Besides, the performance variation is a wide-spread issue across all of the pipelines mentioned in Section 3. And in fact, we divided all the pipeline jobs (around 200K) into 474 comparable groups at the time of writing this paper. Among the largest 20 groups (constituting 36.5% of all the jobs), all of them exhibit a long tail in performance, and 95th percentile performance is larger than 1000 seconds/GB for 12 groups.

To show that the behavior in Figure 7b is not because of degraded machines or hardware, we take six VMs that run on six different machines and show the statistics of the job performance on these VMs in Figure 8 in a boxplot. In every VM (*e.g.*, VM1 on machine1), users can run the same job pipeline repeatedly. Maximum and minimum *processing rate* are represented by the top and bottom bars, and 75th and 25th percentiles are represented by the top and bottom edges of the rectangle box, and median value is represented by the line inside the rectangle box. As shown in the fig-

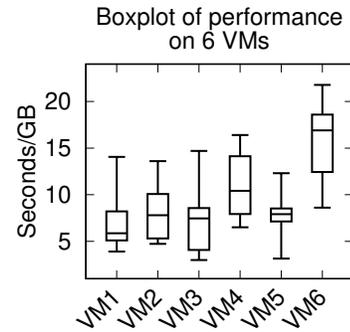


Figure 8: **Performance variance across VMs, §4.1.**

ure, the *processing rate* in a single VM varies (even though the same pipeline) and the performance across the VMs also varies (even though the VMs have the same configuration).

4.2 Application-Level Measurement

We manually noticed that jobs started running slower on VMs that have been running for *several days*. Thus, in order to understand the root cause to the issue, we conducted a set of experiments that included micro-benchmarks and real workloads. All experiments were conducted on a pair of VMs with the same virtual hardware setup and the same software setup, except that one of the VMs was cold-restarted before the experiment. We refer this cold-restarted VM as a *Fresh VM*. The other VMs (*Aged VM*) had been running for a few days and were selected for close monitoring after slow jobs were observed. Each VM was the only tenant on its host and configured to have 40 vCPUs, 226GB RAM and 2.5 TB storage. The hosts were equipped with two 2.20GHz 12-core 24-thread Intel Processors Xeon E5-2650 v4, 504GB RAM and 7.3 TB SSD RAID-5 storage. Linux-4.4 kernel, libvirt 1.3.1 and OpenStack Nova 13.1.4 were installed for virtual machine support.

First, we ran a simple application to reproduce our observation about a *Fresh VM* vs an *Aged VM*. To simplify the experiment, we broke down a widely used pipeline in GPAS (Somatic Variant Calling) and only selected one of the tools, VarScan2 [21]. Job pipelines in GPAS are spawned as multiple processes, hence VarScan2 can run as multiple processes. The details and the reason for parallelism is explained in [10].

Since the input data for all the tests in this experiment are the same, here we do not use *processing rate*, but instead *execution time* as the performance metric. A higher execution time implies worse performance (the same as in *processing rate*).

We define an “*n*-process VarScan2 task” as a task that uses *n* VarScan2 processes to process the input data. We define a “*test*” as an experiment that concurrently runs 5 tasks of the 8-process VarScan2 on a single VM. The input data is replicated 5 times and each task performs the same computation on the same content but distinct replicas of the data. After a test completes, we measure and record the *average execution*

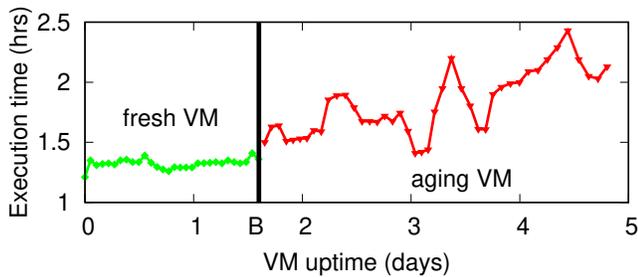


Figure 9: **VarScan2 experiment, §4.2.** Each point in the figure represents a test of five 8-process VarScan2 tasks.

time of the 5 tasks in the test and then repeat the test until 5 days have elapsed.

Figure 9 shows the average execution time of the tasks across several days. Every y point represents a test; the y value of a point shows the average execution time of the 5 concurrent tasks in the test. We can observe that between day 0 and 1.6 (marked by line B), the execution time of the test is relatively fast with low variance. However, after approximately 1.6 days, the performance starts to degrade. We observe that VarScan2 tasks perform normally on a *Fresh VM*, but perform much worse on an *Aged VM*.

4.3 Kernel-Level Measurement

To understand the slow performance in an *Aged VM*, we conducted further experiments including micro-benchmarks and in-kernel measurements in the *Aged VM*.

We use `sysbench` [29], a configurable multi-threaded benchmark tool that provides a variety of tests for benchmarking CPUs, multi threading, memory operation, and file I/O performance. We performed many varieties of experiments (not shown here for space) and found that most benchmarking results do not reveal much difference between an *Aged VM* and a *Fresh VM* (not shown), except for file I/O performance. Not only does the I/O throughput show different results; but, more interestingly, the monitored CPU utilization on *Aged VM* and *Fresh VM* are quite different when compared to that of the host OS.

A common way to monitor CPU utilization is calculating the difference of accumulated values in the pseudo file system (`/proc/stat`). Simply speaking, the values represent how many time slices have been used for each type (user, system, etc) and for each CPU. There are many tools that profile CPU utilization including `top` and `scolllector`. We use the latter as it collects and saves raw data, and we can use other tools to calculate and visualize it in desired ways.

Figure 10 shows the comparison of CPU utilization collected from the host OS versus inside an *Aged VM*. In the VM, we ran a `sysbench` file I/O test (on an SSD). It is worth noting that there are two lines in the figure (`UtilRaw` and `UtilDrv`) representing two ways we calculate CPU utilization. `UtilRaw` is the raw CPU utilization number (in %)

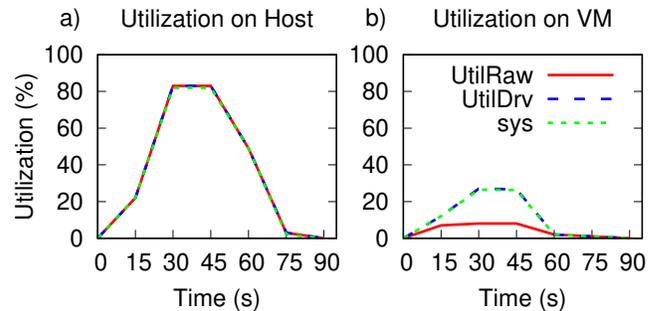


Figure 10: **CPU utilization of sysbench, §4.3.** CPU utilization during `sysbench` file I/O test is presented in this figure. The graph to the left shows CPU utilization collected from the host OS, while the one to the right is collected from an Aged VM. The VM is the only tenant on its host.

that `scolllector` outputs in every second. In addition to the raw CPU utilization, `scolllector` also outputs more detailed information, including `cpuUserSlices`, `cpuSysSlices`, and `cpuIdleSlices`. We define `UtilDrv` (for derived) by summing the user and system time slices and dividing by all slices ($(\text{cpuUserSlices} + \text{cpuSysSlices}) / \text{allSlices}$). Again the difference is that `UtilDrv` simply sums the `cpuUserSlices` and `cpuSysSlices` without considering the `idleSlices`. The figure also shows another line `sys`, which represents `cpuSysSlices` divided by all the slices (in %).

We make the following important observations: (a) System (`sys`) CPU utilization is high on the host and equal to the overall CPU utilization. A similar observation can be found in the *Aged VM*. We consider this abnormal because the workload is I/O bound. (b) At the peak utilization, the CPU utilization observed on the host is 82%, while it is only 27% on the *Aged VM*. Note again that there is only one VM on the host and no other heavy workloads running on the host. This implies that *the hypervisor works intensively, a hidden CPU overhead*. (c) On the VM, there is a gap between the two ways we measure CPU utilization (the gap between `UtilDrv` and `UtilRaw`). Normally, these two lines should overlap as in the host-level measurement (left graph). What happens here is that `scolllector` assumes the VM always gets all the CPU slices from the host OS. However, with our method, `UtilDrv`, it shows there is a “loss of time” due to the hidden overhead in the hypervisor. CPU time that is supposed to be used for tasks in the VM was used for other system tasks in the host. (d) On a separate measurement on a *Fresh VM* (not shown for space), we found no such high system CPU usage nor a gap between the two lines.

4.4 Memory Fragmentation

A major problem of aging resources is fragmentation. We started suspecting there was a memory fragmentation problem where sequential guest pages were not mapped sequentially on the physical pages. To get more evidence, we ran concurrent processes and analyzed read operations. Roughly

Tasks	1 × 1	1 × 8	4 × 8	5 × 8
Steps #1-4	< 1	< 2	< 12	< 20
Step #5	5	92	290	512
Total	6	94	306	552

Table 4: **Read latency break-down, §4.4.** Average total time (in seconds) that each process spends in the five steps during read operations are listed in the table. $x \times n$ at the top row means a test with x number of n -process VarScan2 tasks.

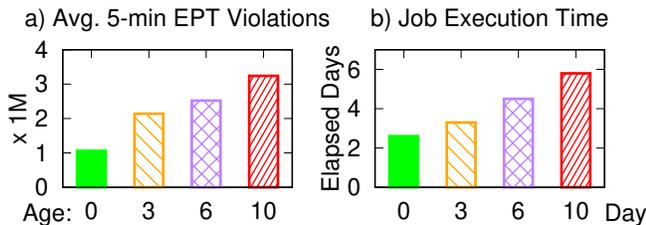


Figure 11: **Correlations between EPT violations and job execution time, §4.5.** Each bar represents the statistics for jobs when the VM is 0, 3, 6, and 10 days old. EPT violations are recorded by enabling tracing on the physical host.

speaking, a file read operation goes through five steps: 1) check the page cache to see whether the data is already in memory; 2) conduct a synchronized read request at the file system level; 3) send out an asynchronized read request at the block level; 4) wait for completion; and, 5) copy the data from the kernel to the user space. Step (5) represents the most memory-intensive step among all the steps.

We conducted an experiment that records the time spent in each of the steps. We set up four tests. The first test runs a single 1-process VarScan2 task, and the other four tests run 1, 4, and 5 (concurrent) 8-process tasks, respectively.

Table 4 shows the average time every process spends on each I/O step. Note that most of the time is spent in Step #5 (memory copying). With just 1 process (1x1), a process only takes 5 seconds in total for memory copying. With 40 processes (5x8), every process now takes 512 seconds in step #5, which is roughly a $100x$ slowdown. We note that there is other contention in the SSD (as can be seen in steps #1-4); but, even with 40 processes, the I/O waiting time is not as severe as the slowdown from memory-copying. We also note that we have a 48-core machine, hence CPU contention should be almost negligible with 40 processes.

4.5 The Root Cause: EPT Violation

In this section, we quantify the root cause. After considerable investigation, we found that the root cause resides in the *extended page table (EPT)*, which is a technology invented to increase virtual memory performance for VMs. The use of EPT by the hypervisor is designed to be transparent to VM users. To our knowledge, EPT is used by only certain hypervisor implementations, such as Linux KVM. In a nutshell, EPT serves as a page table that stores the mapping between

the VM memory address and the host physical memory address. Modern CPUs use the translation look-aside buffer (TLB) to store a small subset of the EPT entries. Whenever there is a TLB miss, an *EPT violation* occurs, which causes the hypervisor to interrupt the VM to handle the violation.

Figure 11 shows an experiment with 5 jobs using 30 cores running repeatedly on a fresh state VM for ten days. Figure 11a shows the average number of EPT violations (in millions) observed in every 5 minutes. The figure clearly shows that the longer the VM has been running, the higher the number of EPT violations. Figure 11b confirms the correlation between the number of EPT violations and job execution time.

In conclusion, VM aging leads to more frequent EPT violations causing the hypervisor to interrupt the VM more frequently. In the next two sections, we describe how we monitor and mitigate the problem.

5 Performance Management

To manage this performance problem, we discuss the two parts of our solution: monitoring and mitigation.

5.1 Monitoring

We suggest two methods to detect VM aging: monitoring EPT violation (in the host) or CPU utilization gap (in the VM), along with with the challenges.

Host-Level EPT Violation Monitoring One direct way to measure the problem is to count the number of EPT violations observed in the hypervisor, however the result is relative—how do we know whether the number represents a higher than normal number of EPT violations. We found another more concrete metric to measure this problem: *address distance of subsequent EPT violations* (which basically attempts to measure the level of memory fragmentation). For example, if two subsequent violations at time T and $T+t$ are about translation misses of guest pages #100 and #2000, respectively, then the distance recorded is $1900 \times 4\text{KB}$. Essentially, we argue that when the addresses of subsequent EPT violations are farther apart, the memory tends to be more fragmented and more EPT violations will occur, causing more time to be spent managing EPT violations.

We return to the experiment in Section 4.5 and this time plot the distribution of address distances of subsequent EPT violations. Figure 12 shows different distributions categorized based on the age of the VM age. Notice that older VMs have distinctly larger address distances. For example, in a 10-day old VM, we can see a distance of at least 50GB in roughly 40% of the time ($x=50\text{GB}$, $y=0.6$).

This method requires access to the host. It can provide performance alerts in advance. For example, the distance distribution in a 3-day old VM can be clearly distinguished from a fresh VM. Here, the resulting job execution time has

Application / VM	vCPU Efficiency (%)	Execution Time
Heavy / Fresh VM	99	16.0 hrs
Heavy / Aged VM	83	39.0 hrs
Light / Fresh VM	99	5.4 hrs
Light / Aged VM	99	5.6 hrs

Table 5: **vCPU Efficiency, §5.1.** vCPU Efficiency and execution time for applications on the Fresh VM and Aged VM are listed in the table. There only shows a difference in vCPU Efficiency for the “heavy” application.

been increased by 27% (which was hard to observed in the middle of the job). Thus, this kind of monitoring allows us to predict job performance degradation even before the job ends. In terms of performance overhead, sampling violations in an online manner may bring an impact to system performance. However, our experiment where the trace is enabled for five minutes every ten minutes does not show a negative impact on performance.

VM-Level /proc/stat Monitoring

While the above method requires host-level access, we now present another method that can be done at the VM level. As explained before, the Linux kernel implements the `proc` pseudo-filesystem which provides an interface to kernel data structures. The `/proc/stat` file provides time-slice statistics across user, system, and idle processes from the time the system boots up. In our deployment, the Linux time slice is configured to 10ms.

Inspired by the “gap” shown in Figure 10 earlier, it is possible for users to monitor the gap at the VM level. *The gap is caused by a phenomenon where a time slice in the VM is actually (and significantly) less than 10ms because the hypervisor is handling EPT violations.* Thus, we can introduce a simple metric, *vCPU Efficiency*, which is the sum of all the time-slice values in `/proc/stat` (user, sys, and idle values) divided by the real time slices that have elapsed (since the last time we read `/proc/stat`). The metric should be near 100%, but when EPT violation is high, it is expected that the efficiency will be much lower than 100%.

Table 5 shows *vCPU Efficiency* and the job execution time for the same experiments we ran before. Here, we select one “heavy” and one “light” application, where the heavy application uses all 6 available cores per job and the light application uses 1 core per job. We can see that for heavy workloads, there is a large difference in *vCPU Efficiency* and

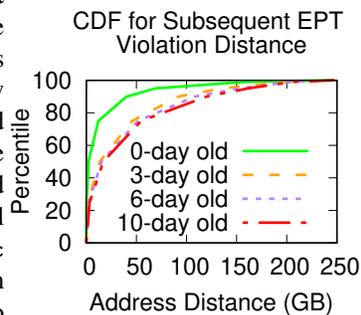


Figure 12: **EPT violation distance CDF, §5.1.**

Mitigation	Pros	Cons
Using Huge Pages	Performance is close to bare metal nodes.	Huge page VMs are more complicated to configure, and less flexible. Benefits of huge pages could be offset with even larger memory size and memory usage.
Restarting VM	Performance is best after restarting.	The system must support job check-pointing. Longer down time.
Defragmenting Memory	Performance is improved, short down time.	Cannot provide the best performance. Improvement is only temporary.
Running on bare metal	Performance is best and sustainable.	Does not provide the flexibility, management advantages, nor security isolation of VMs
Using public clouds	Based on one-week experiments, the performance is best and sustainable. Easy to manage.	higher costs for some workloads.

Table 6: **Pros and cons of five mitigation methods, §5.2.**

execution time. For example, when *vCPU Efficiency* drops from 99% to 83%, the resulting job execution time increases from 16 to 39 hours.

The disadvantage of this method is that we cannot detect VM aging unless we run a heavy application that can be impacted by the aging. Table 5 shows that for a light application, there is no visible difference in the *vCPU Efficiency* and execution time, even though the VM is already degraded at the time of running. We also want to emphasize that VM aging *cannot* be crudely defined by the number of days a VM has been up running. In our deployment, VMs age fast (after 3-6 days) because we ran complex bioinformatics pipelines.

5.2 Mitigations

This section describes several ways that we tried to address the problem. The choice of which mitigation to use depends upon the system requirements, and system administrators will need to decide which mitigation technique works best for them. Table 6 summarizes the pros and cons of the mitigation techniques we discuss below.

Using Huge Pages Just like a standard page table, the EPT size increases as the memory grows larger. Also the smaller the page size, the higher the probability of misses. Increasing the page size to 1MB for example (*i.e.* using “huge” pages) shrinks the size of the EPT table and reduces the probability

Restart Strategy	Jobs per day
Proactive restart	1.92
Slowdown-triggered restart	1.69
No restart	1.23

Table 7: **Rebooting VM mitigation, §5.2.**

of misses. However, in GPAS this is not an easy option to adopt. Huge page VMs are less flexible. Configuring and debugging huge page VMs in a production environment takes time. Hence, it is not easy to reconfigure and troubleshoot all the machines with a huge page configuration. In addition, a huge page size that is “huge” enough for now is not a permanent solution given the growing sizes of memory and the increasing application memory usage expected in the future [30]. Another option is to keep the 4KB page unit, but allocate smaller VMs to reduce memory fragmentation. However, in GPAS, resource requirements differ across jobs, and many jobs require large memory VMs.

Restarting VMs to avoid performance degradation Another method is to restart the VMs occasionally to “reset” the memory fragmentation. According to Figure 11, it is possible to detect EPT violations early before performance degrades significantly. With such a detection, we can decide when is a proper time to restart. We conducted a simple experiment with the same jobs as in Figure 11. Table 7 shows that restarting increase the number of jobs finished per day. Here, “proactive restart” implies restarting the VM after every job finishes (*i.e.*, do not reuse the VM across jobs) and “slowdown-triggered restart” implies restarting when the monitored *vCPU Efficiency* drops below 90%. The throughput numbers in Table 7 might also suggest that restarting in the middle of a long job might improve its execution time. However, this requires checkpointing the job progress, which a feature that is not supported in GPAS.

Defragmenting Memory The burst of EPT violations in aging VMs is essentially caused by the loss of data locality of the VM memory on the host physical memory. We can use the built-in memory defragmentation tool in Linux to reorganize the memory layout. A simple experiment with VarScan2 workloads shows that defragmentation indeed helps decrease EPT violations. As shown in Table 8, the test runs 21% faster after defragmentation when the VM has been heavily used for 7 days. The number of EPT violations during the test is decreased by 58%. However, comparing to performance of the fresh state, this method is still 44% slower and EPT violations are nearly 100 times more. In other words, memory defragmentation can be a temporary method to improve performance by a small margin, but restarting VMs leads to a better outcome.

Running on Bare Metal In GPAS, the most viable alternative is to run jobs directly on bare metal nodes without

VM age	Exec. Time (s)	EPT violations
0 day	587	630,636
7 days	1,073	140,677,142
7 days, defragmented	847	58,607,955

Table 8: **Memory defragmentation, §5.2.**

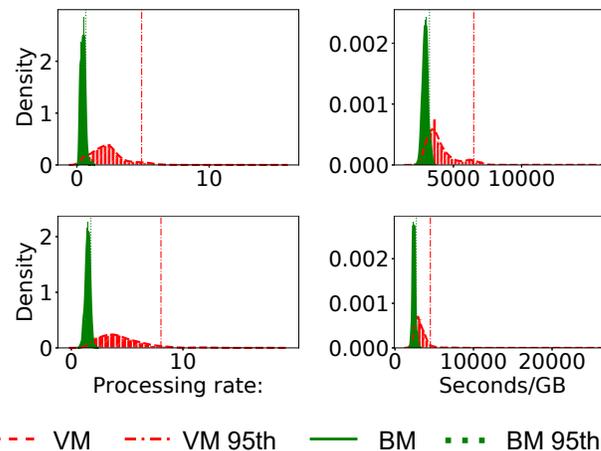


Figure 13: **Gaussian density estimation on processing rate, §5.2.** Gaussian density estimation has been applied to processing rate of jobs for four pipelines. Histograms are also shown for each pipelines. Vertical lines show where the 95th percentile processing rate resides for bare metal nodes and VMs.

VMs. The caveat is that not all research projects (jobs) can run in this mode; some research projects require the security isolation provided by running the jobs in a secure VM. For this reason, GPAS now uses a combination of bare-metal and VM deployments. The statistics presented below are from job pipelines that have more than one hundred jobs on bare metal nodes.

In Figure 13, a Gaussian kernel density estimation is calculated for jobs of four particular pipelines on bare metal (BM) nodes and VMs. Bare-metal jobs exhibit much less variance than VM jobs (the bell shapes of bare-metal jobs are more localized across the x-axis). In addition, bare-metal jobs have higher *processing rate* than VM jobs. The 95th percentile *processing rate* is marked in the graphs. Note that the *processing rate* tail of the bare metal jobs is relatively short.

Table 9 shows the performance improvements of using bare metal vs VMs for the fifteen GPAS pipelines that had at least 100 or more bare metal jobs. The *processing rate* improvement (in %) at is shown in the table for different percentiles. For example, the 95th percentile *processing rate* is improved by between 22% and 95%. The average performance improvement is between 18% and 87% (not shown).

In summary, given these benefits, the GDC platform uses a combination of bare metal and VM nodes, which provides good performance, but more complexity when scheduling jobs and more overhead when managing nodes.

Using Public Clouds We next turn to the question of whether there are problems with long running bioinformatics

Percentile	Improvement	Percentile	Improvement
25 th	14 – 78	95 th	22 – 95
median	18 – 79	97 th	20 – 97
75 th	19 – 81	max	28 – 100

Table 9: **Running on bare metal, §5.2.** *Processing rate improvements on different percentiles by running jobs on bare metal nodes are listed in the table.*

pipelines on the virtualization stacks used in public clouds. Different virtualization stacks are likely to use different approaches, which may, or may not, have the same issues with EPT violations. Some vendors [31, 32] recommend in their documentation that VMs with large memories use configurations with huge page memory, but with no more specific guidance given. On the other hand, hypervisors such as Xen, use an approach to implement virtual machine memory management that directly maps guest virtual address to host physical address (called direct paging). For this reason, they do not incur any additional overhead when resolving the mapping from guest to host, as KVM does.

We did experiments using Amazon Web Services (AWS) and Google Cloud Platform (GCP) to understand whether there are any problems with long running bioinformatics pipelines in their virtualization stacks. To simplify the experiments, we ran VarScan2 using open access data so that the full security and compliance infrastructure normally required by GPAS would not have to be used.

Amazon Web Services has developed their own Nitro system based on KVM. We rented a dedicated host (z1d) and allocated a large memory VM (12xlarge). The one-week experiment does *not* show performance degradation. For the dataset we used, the minimum execution time is 3.6 hours and the maximum is 3.9 hours, as shown in Table 10. Nitro may offload many tradition virtualization functions to dedicated hardware, which we suspect avoids the address translation overhead.

Google Cloud Platform also developed their hypervisor based on KVM [33]. We rented a 96-core sole-tenant host to avoid sharing with other users and allocated a 90-core 576GB VM on the host. We repeated the same experiment while scaling the maximum number of jobs to 14. The experiment results also does *not* show any degradation either. The minimum and maximum execution times are 5.7 and 6.0 hours, respectively. Although they use KVM, the same problem might not appear due to one of the following potential reasons: they use huge pages; they use software-based memory management instead of EPT; or the CPUs they use have larger TLBs.

In summary, public cloud platforms do not have the problems with performance degradation for long running pipelines that we observed with our on-premise Open-Stack/KVM platform. On the other hand, as mentioned, the

	Tests	Min (hrs)	Max (hrs)
One on-prem VM	36	10	15.3
Amazon Web Services	69	3.6	3.9
Google Cloud Platform	98	5.7	6.0

Table 10: **DNA alignment workload execution time on public cloud, §5.2.**

large scale GPAS on-premise system has lower costs than public clouds for many of the GDC workloads.

For all of the reasons stated in this section, the GDC platform uses a combination of VM and bare-metal, with occasional VM restarts. The GDC also uses public clouds for some workloads, to provide flexibility, and for burst computing. In summary the GDC today uses a hybrid on-premise/public cloud to take advantage of the flexibility of public clouds and the lower costs provided by large scale on-premise clouds for some workloads.

6 Future Challenges

After addressing the EPT violation problem, our future goal is to improve resource utilization of our cluster. The GPAS job management system runs jobs that are encapsulated as CWL workflows [7] and Dockerized. Data that is submitted to the GDC are organized into projects or portions of a project called batches. First, GPAS must schedule different competing projects/batches. For simplicity, we will just describe the process for scheduling projects, since scheduling batches is similar but more complicated. Projects typically contain multiple data types and the appropriate CWL pipelines must be run over each data type. In the first step, GPAS schedules the running of the required CWL pipelines over the different data types in each project. In the second step, CWLtool processes the DAG in the CWL pipeline and schedules each step in the DAG. In the third step, SLURM manages the running of jobs submitted by CWLtool on the VM pool of the available compute nodes.

Currently, each bioinformatics pipeline is encapsulated in a single CWL workflow that is containerized, and the container is scheduled and assigned to a virtual machine for execution. The pipelines were developed by the research community over a period of time and some employ threads efficiently, while others are less efficient. Since portions of pipelines may be either CPU-bound or I/O bound, inefficiencies can arise. For example, Figure 14 shows five Somatic-Variant-Calling (I/O intensive) jobs consuming different input files. The left figure shows that I/O utilization is full but the total bandwidth is lower than the maximum bandwidth of the hard drive (due to seek contention). The right figure shows that for most of the time, CPU is spent in waiting for I/O. The end part (5PM) of the figure also highlights the problem in the last paragraph where there is low I/O activity but the CPUs are not fully utilized.

	CURRENT:						PROPOSED:					
P1:	A1	A2	A3	B1	B2	B3	A1	A2	A3			
P2:			A3		B3		B1	B2	A3	B3		
P3:										B3		
Time:	1	2	3	4	5	6	1	2	3	4		

Inefficiencies can also arise due to the fact that currently GPAS assigns a VM per CWL container, and different portions of the CWL workflow benefit from different number of processors. As a specific example, assume we have a machine with three processors (P1–P3), and two jobs A and B, each of which has three serialized tasks (1, 2, 3) and task# 3 runs as threads. Assume that the maximum resource that A and B need are two processors, but that these are just needed for a portion of the pipeline. In this case, we have a classical scheduling allocation problem, if we devote two processors for A and B, then there are periods in which the CPUs are under utilized as shown in the Figure (the white spaces).

It would be more efficient to allocate the processors as shown on the right, where we allow both jobs A and B to be assigned on the same machine even though the total processors needed exceeds the number of processors ($4 > 3$). A simple simulation of some jobs in our cluster shows such an approach like this can increase CPU utilization up to 20% and reduce job execution time up to 15%.

While there is a vast literature on job/task scheduling (see Section 7), the challenge here is that the CWL workflows must either be rewritten and an appropriate scheduling algorithm used or the CWL execution engine itself must have a greater ability to parallelize portions of the DAG being executed. Both of these approaches are currently being developed.

7 Related work

We now discuss related work briefly for space (please see our supplemental material for more [10]).

The introduction of EPT is aimed to increase the performance of VM memory management. In its early stage of development, many found it to be effective in reducing the translation overhead compared to software-based solutions [34]. Multiple recent studies [4, 35–37] show that *short-term* benchmarks running on more advanced modern CPUs, the overhead from TLB misses and EPT violations is small and the VM performance is comparable to bare-metal performance. We found this is not true for our large bioinformatics jobs. Huge pages can reduce TLB misses to a large extent [38, 39], but it is not always a viable solution. More recent research in the community [30, 40–44] has devoted themselves to devise a better virtual memory management or mitigate the overhead of TLB misses. Another lesson that can be taken here is the need for tools to quickly reproduce memory (allocation) aging just like the popular utility of filesystem aging tools [45–53].

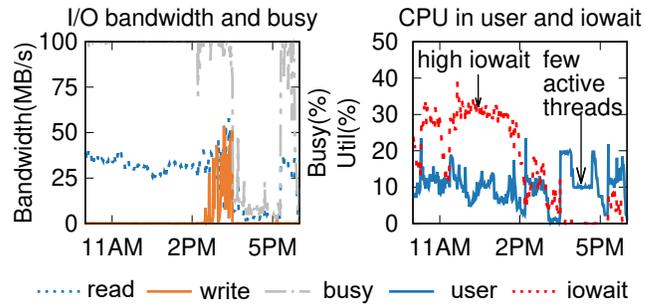


Figure 14: CPU under utilization.

On task scheduling, the literature is also rich on techniques that suggest finer-grained scheduling [54–56], better fairness [57–61], VM/Container backup/migration [62–64], hybrid scheduling [65–67]. The majority of the literature is concerned with optimizing new jobs that are written for a system, while the challenge with GPAS is parallelizing batches of older pipelines, most of which were designed for a single multi-core machine running a single job rather than for processing large datasets of multiple heterogeneous jobs running on a distributed platform.

8 Conclusion

To the best of our knowledge, we are the first to conduct a prolonged performance evaluation of virtualization stack for jobs that are both long running and memory intensive, such as bioinformatics jobs, and hence causing extreme virtual memory fragmentation. Diagnosing this problem has been a long and onerous process, primarily because the problem cannot be quickly reproduced; every experiment must be repeated for days to provide the evidence required. We hope the contributions of this paper can help other deployments similar to ours and lead to new research activities (*e.g.*, memory aging tools).

Acknowledgment

We thank James Bottomley, our shepherd, and the anonymous reviewers for their tremendous feedback and comments.

This project was funded in part with Federal funds from the National Cancer Institute, National Institutes of Health, agreement 14X050 and task order T02 under agreement 17X147 under contract HHSN261200800001E and NSF Grant Numbers CNS-1563956 and CCF-2028427. The content of this publication does not necessarily reflect the views or policies of the Department of Health and Human Services or the National Science Foundation, nor does mention of trade names, commercial products or organizations imply endorsement by the US Government.

References

- [1] Nicholas M Luscombe, Dov Greenbaum, and Mark Gerstein. What is bioinformatics? a proposed definition and overview of the field. *Methods of information in medicine*, 40(04):346–358, 2001.
- [2] K. A. Wetterstrand. Dna sequencing costs: Data from the nhgri genome sequencing program. www.genome.gov/sequencingcostsdata, 2020.
- [3] L Koumakis, C Mizzi, and G Potamias. Bioinformatics tools for data analysis. In *Molecular Diagnostics*, pages 339–351. Elsevier, 2017.
- [4] Timothy Merrifield and H Reza Taheri. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 25–35, 2016.
- [5] Sheng Yang. Extending kvm with new intel virtualization technology. In *KVM forum*, 2008.
- [6] John Vivian, Arjun Arkal Rao, Frank Austin Nothhaft, Christopher Ketchum, Joel Armstrong, Adam Novak, Jacob Pfeil, Jake Narkizian, Alden D Deran, Audrey Musselman-Brown, et al. Toil enables reproducible, open source, big biomedical data analyses. *Nature biotechnology*, 35(4):314–316, 2017.
- [7] Peter Amstutz, Michael R Crusoe, Nebojša Tijanić, Brad Chapman, John Chilton, Michael Heuer, Andrey Kartashov, Dan Leehr, Hervé Ménager, Maya Nedeljkovich, et al. Common workflow language, v1. 0. 2016.
- [8] Workflow description language specification. <https://software.broadinstitute.org/wdl/documentation/spec>, 2019.
- [9] Cwltool repository. <https://github.com/common-workflow-language/cwltool>, 2020.
- [10] Extended report, including extended background, related work, explanations and code. <https://tinyurl.com/biosys-tr>, 2020.
- [11] National cancer institute gdc documentation. <https://docs.gdc.cancer.gov/>, 2021.
- [12] Overview of gdc harmonization workflows. <https://github.com/NCI-GDC/gdc-workflow-overview>, 2020.
- [13] Heng Li and Richard Durbin. Fast and accurate long-read alignment with burrows–wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.
- [14] G Tischler. biobambam2. <https://github.com/gt1/biobambam2>, 2017.
- [15] Picard toolkit. <http://broadinstitute.github.io/picard/>, 2019.
- [16] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, et al. The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. *Genome research*, 20(9):1297–1303, 2010.
- [17] Alexander Dobin, Carrie A Davis, Felix Schlesinger, Jorg Drenkow, Chris Zaleski, Sonali Jha, Philippe Batut, Mark Chaisson, and Thomas R Gingeras. Star: ultrafast universal rna-seq aligner. *Bioinformatics*, 29(1):15–21, 2013.
- [18] Gdc htseq expression quantification workflow. <https://github.com/NCI-GDC/htseq-cwl>, 2020.
- [19] Yu Fan, Liu Xi, Daniel ST Hughes, Jianjun Zhang, Jianhua Zhang, P Andrew Futreal, David A Wheeler, and Wenyi Wang. Muse: accounting for tumor heterogeneity using a sample-specific error model improves sensitivity and specificity in mutation calling from sequencing data. *Genome biology*, 17(1):178, 2016.
- [20] David E Larson, Christopher C Harris, Ken Chen, Daniel C Koboldt, Travis E Abbott, David J Dooling, Timothy J Ley, Elaine R Mardis, Richard K Wilson, and Li Ding. Somaticsniper: identification of somatic point mutations in whole genome sequencing data. *Bioinformatics*, 28(3):311–317, 2011.
- [21] Daniel C Koboldt, Qunyuan Zhang, David E Larson, Dong Shen, Michael D McLellan, Ling Lin, Christopher A Miller, Elaine R Mardis, Li Ding, and Richard K Wilson. Varscan 2: somatic mutation and copy number alteration discovery in cancer by exome sequencing. *Genome research*, 22(3):568–576, 2012.
- [22] Gatk mutect2. <https://gatk.broadinstitute.org/hc/en-us/articles/360037593851-Mutect2>, 2019.
- [23] dockstore-cgpwgs. <https://github.com/cancerit/dockstore-cgpwgs>, 2020.
- [24] Gdc vep annotation workflow. <https://github.com/NCI-GDC/vep-cwl>, 2020.
- [25] Maf-lib: the mutation annotation format library. <https://github.com/NCI-GDC/maf-lib>, 2020.
- [26] snp6cbs: Segment tcga snp6 tangent normalized data. <https://github.com/NCI-GDC/dnacopy-tool>, 2020.
- [27] Simon Andrews et al. Fastqc: a quality control tool for high throughput sequence data, 2010.
- [28] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, and Richard Durbin. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078–2079, 2009.
- [29] Alexey Kopytov. Sysbench: a system performance benchmark. <http://sysbench.sourceforge.net/>, 2004.
- [30] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. Efficient virtual memory for big memory servers. *ACM SIGARCH Computer Architecture News*, 41(3):237–248, 2013.
- [31] Huge pages in vmware vcloud nfv openstack edition. <https://docs.vmware.com/en/VMware-vCloud-NFV-OpenStack-Edition/3.0/vmwa-vcloud-nfv30-performance-tuning/GUID-1F05987F-012B-4BC4-9015-CDE3C991C68C.html>, 2018.
- [32] How is the hugepages feature enabled for virtual machines? https://docs.oracle.com/cd/E64076_01/E64081/html/vmcon-vm-hugepages.html, 2018.

- [33] Google compute engine faq. <https://cloud.google.com/compute/docs/faq>, 2020.
- [34] Nikhil Bhatia. Performance evaluation of intel ept hardware assist. *VMware, Inc*, 2009.
- [35] Tom Spink, Harry Wagstaff, and Björn Franke. Hardware-accelerated cross-architecture full-system virtualization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(4):1–25, 2016.
- [36] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.
- [37] Jeffrey Buell, Daniel Hecht, Jin Heo, Kalyan Saladi, and R Taheri. Methodology for performance analysis of vmware vsphere under tier-1 applications. *VMware Technical Journal*, 2(1):19–28, 2013.
- [38] Jerry Huck and Jim Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th annual international symposium on computer architecture*, pages 39–50, 1993.
- [39] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *ACM SIGOPS Operating Systems Review*, 36(SI):89–104, 2002.
- [40] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 705–721, 2016.
- [41] Ashish Panwar, Aravinda Prasad, and K Gopinath. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 679–692, 2018.
- [42] Jayneel Gandhi, Christopher J Rossbach, and Timothy Merrifield. Decoupling memory metadata granularity from page size, September 12 2019. US Patent App. 15/916,173.
- [43] Jayneel Gandhi, Arkaprava Basu, Mark D Hill, and Michael M Swift. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 178–189. IEEE, 2014.
- [44] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, and John CS Lui. Smartmd: A high performance deduplication engine with mixed pages. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 733–744, 2017.
- [45] 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 *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 691–704, 2018.
- [46] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A Bender, William Jannen, Rob Johnson, Donald Porter, and Martin Farach-Colton. Filesystem aging: It’s more usage than fullness. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.
- [47] Alex Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A Bender, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, et al. File systems fated for senescence? nonsense, says science! In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 45–58, 2017.
- [48] 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, 2009.
- [49] Michael P Mesnier, Matthew Wachs, Raja R Simbasivan, Julio Lopez, James Hendricks, Gregory R Ganger, and David R O’hallaron. //trace: parallel trace replay with approximate causal events. 2007.
- [50] Akshat Aranya, Charles P Wright, and Erez Zadok. Tracefs: A file system to trace them all. In *FAST*, pages 129–145, 2004.
- [51] Alan D Brunelle. Block i/o layer tracing: blktrace. *HP, Gelato-Cupertino, CA, USA*, 57, 2006.
- [52] Nikolai Joukov, Timothy Wong, and Erez Zadok. Accurate and efficient replaying of file system traces. In *FAST*, volume 5, pages 25–25, 2005.
- [53] Zev Weiss, Tyler Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Root: Replaying multithreaded traces with resource-oriented ordering. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 373–387, 2013.
- [54] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 184–200, 2017.
- [55] Tatiana Jin, Zhenkun Cai, Boyang Li, Chengguang Zheng, Guanxian Jiang, and James Cheng. Improving resource utilization by timely fine-grained scheduling. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.
- [56] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 81–97, 2016.
- [57] 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.
- [58] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 65–80, 2016.
- [59] Călin Iorgulescu, Florin Dinu, Aunn Raza, Wajih Ul Hassan, and Willy Zwaenepoel. Don’t cry over spilled records: Memory elasticity of data-parallel applications and its

- application to cluster scheduling. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 97–109, 2017.
- [60] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Nsdi*, volume 11, pages 24–24, 2011.
- [61] Matei Zaharia, Dhruva Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278, 2010.
- [62] 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, 2017.
- [63] Cheng Wang, Bhuvan Urgaonkar, Aayush Gupta, George Kesidis, and Qianlin Liang. Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 620–634, 2017.
- [64] Daeyong Jung, SungHo Chin, Kwang Sik Chung, and HeonChang Yu. Vm migration for fault tolerance in spot instance based cloud computing. In *International Conference on Grid and Pervasive Computing*, pages 142–151. Springer, 2013.
- [65] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 499–510, 2015.
- [66] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–13, 2018.
- [67] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 485–497, 2015.

Scaling Large Production Clusters with Partitioned Synchronization

Yihui Feng*
Alibaba Group

Zhi Liu*†
The Chinese University of Hong Kong

Yunjian Zhao†
The Chinese University of Hong Kong

Tatiana Jin†
The Chinese University of Hong Kong

Yidi Wu†
The Chinese University of Hong Kong

Yang Zhang
Alibaba Group

James Cheng†
The Chinese University of Hong Kong

Chao Li
Alibaba Group

Tao Guan
Alibaba Group

Abstract

The scale of computer clusters has grown significantly in recent years. Today, a cluster may have 100 thousand machines and execute billions of tasks, especially short tasks, each day. As a result, the scheduler, which manages resource utilization in a cluster, also needs to be upgraded to work at a much larger scale. However, upgrading the scheduler — a central system component — in a large production cluster is a daunting task as we need to ensure the cluster’s stability and robustness, e.g., user transparency should be guaranteed, and other cluster components and the existing scheduling policies need to remain unchanged. We investigated existing scheduler designs and found that most cannot handle the scale of our production clusters or may endanger their robustness. We analyzed one most suitable design that follows a shared-state architecture, and its limitations led us to a fine-grained staleness-aware state sharing design, called partitioned synchronization (ParSync). ParSync features the simplicity required for maintaining the robustness of a production cluster, while achieving high scheduling efficiency and quality in scaling. ParSync has been deployed and is running stably in our production clusters.

1 Introduction

In Alibaba, we operate large clusters each containing tens of thousands of machines. Every day, billions of tasks are submitted to, scheduled and executed in a cluster. A *cluster scheduler*, or *scheduler* for short, manages both machines and tasks in a cluster. Based on the resource requirements (e.g., CPU, memory, network bandwidth) of tasks, a scheduler matches tasks to appropriate resources using various scheduling algorithms and makes complex trade-offs among multiple scheduling objectives such as scheduling efficiency, scheduling quality, resource utilization, fairness and priority of tasks.

The ability to balance these objectives largely depends on both technical and business factors, and thus varies from company to company and cluster to cluster.

Due to the rapid growth in our businesses in recent years, we have faced serious challenges in scaling our scheduler, which is a centralized architecture similar to YARN [44], as there have been substantially more tasks and machines in our clusters. Today, the size of some of our clusters is close to 100k machines and the average task submission rate is about 40k tasks/sec (and considerably higher in some months). This scale simply exceeds a single scheduler’s capacity and an upgrade to a distributed scheduler architecture is inevitable.

In this paper, we present *a design for a distributed scheduler architecture that can handle the scale of our cluster size and task submission rate, while at the same time achieving low-latency and high-quality scheduling*. The design presented also satisfies two hard constraints (§2): *backward compatibility* and *seamless user transparency*. The scheduler’s robustness and stability derived from these constraints are of vital importance for our production environment.

Cluster schedulers have been extensively studied [9, 13, 14, 17, 25, 30, 32, 36, 37, 39, 44, 45] and we discuss the suitability of existing schedulers for our cluster environment and workloads in §3. Among them, a *shared-state scheduler architecture*, described in *Omega* [39], is able to handle our cluster size and satisfy the two hard constraints because it requires minimal intrusive architectural changes. In *Omega*, a master maintains the **cluster state**, which indicates the availability of resources in each machine in the cluster. There are multiple schedulers, each of which maintains a local copy of the cluster state by synchronizing with the master copy *periodically*¹. Each scheduler schedules tasks *optimistically* based on the (possibly stale) local state and sends resource requests to the

¹Omega [39] assumes that there is no synchronization overhead and thus each scheduler synchronizes the entire local state with the master whenever it communicates with the master to commit a task. But the synchronization overhead is not negligible in our cluster as it has a large state (due to the large size of the cluster), which does not allow us to synchronize the state at every task commit because of the high task submission rate (much higher than those in [39]). Thus, we synchronize the state periodically.

*Co-first-authors ordered alphabetically.

†This work was done when the authors were visiting Alibaba.

master. As multiple schedulers may request the same piece of resource, this results in **scheduling conflicts**. The master grants the resource to the first scheduler that requested it, and the other schedulers will need to reschedule their task. The scheduling conflicts and rescheduling overheads lead to high latency when the task submission rate is high, which we validate in §4 using both analytical models and simulations. Our results show that the *contention on high-quality resources* and the *staleness of local states* are the two main contributors to high scheduling latency as they both substantially increase scheduling conflicts.

Then in §5, we propose **partitioned synchronization (ParSync)** to mitigate the negative impacts of these two factors. ParSync partitions the cluster state into N parts (N is the number of schedulers), such that each scheduler synchronizes $1/N$ of distinct partitions instead of the entire state each time. As a result, at any time each scheduler has a fresh view of $1/N$ of the partitions and can first schedule its tasks to these partitions. This significantly reduces the contention (with other schedulers) on high-quality resources. Based on when a partition is synchronized, a scheduler knows how stale its partitions are; thus, algorithm designers can now better balance scheduling latency and scheduling quality by adjusting the preference to fresher partitions. We also devise an adaptive strategy to dynamically choose between lowering scheduling latency and improving scheduling quality. We validate the effectiveness of ParSync and various scheduling algorithms developed based on ParSync in §6. ParSync has been deployed in Fuxi 2.0, which is the latest version of the distributed cluster scheduler in Alibaba, and is running stably in our production clusters, where each production cluster contains thousands to 100K machines.

2 Background and Challenges

Workload statistics. Millions of jobs are submitted to a cluster each day in Alibaba. Figure 1a (solid curve) plots the number of jobs processed in a cluster each day in a randomly picked month, which ranges from 3.34 to 4.36 million jobs. A job consists of many tasks and Figure 1a (dotted curve) shows that the number of tasks each day ranges from 3.1 to 4.4 billion. The majority of tasks are short tasks. As shown in Figure 1b, 87% of tasks are completed in less than 10 seconds.

Motivation for scheduler upgrade. Our previous cluster scheduler follows a typical master-worker architecture [44]. A single *master* manages all the resources in a cluster and handles all the scheduling work. Each worker machine has an *agent* process, which sends the latest status of the worker via heart-beat messages to the master. The master receives jobs submitted by users and then places each job in its corresponding *quota group* [26]². The cluster operator configures a

²Jobs belong to projects and projects are assigned resource quotas according to their budgets. A *quota group* can be considered as a virtual cluster

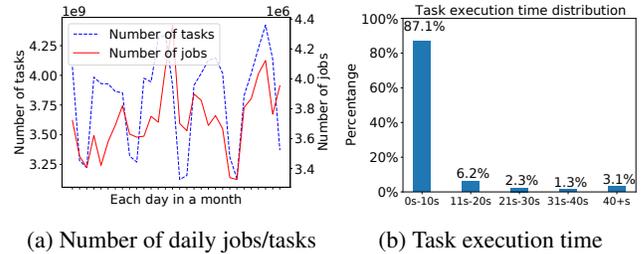


Figure 1: Job/task statistics

quota group to specify the minimum and maximum amounts of resources the group can acquire. In particular, when the cluster is overloaded, resources have to be divided among all the groups by weighted fairness (based on their quotas).

In recent years, the scale of our cluster has significantly increased and a single cluster can have 100k machines. *Statically partitioning a large cluster is not an option* because there are some extremely large jobs from critical projects and a large cluster is required to ensure that these large jobs and a large number of daily production jobs can be both processed without extended delay. There are also important technical reasons (e.g., resource fragmentation [45], limited visibility [39]) and business factors (e.g., projects need to access the data of other projects in the same business unit that stores its data in a single cluster for operational and management reasons), *which require scheduling over an entire large cluster rather than breaking it down into smaller ones*.

However, the previous monolithic single-master architecture could not handle the scale of the current clusters in Alibaba. First, the 10x larger number of machines requires a larger time gap between two consecutive heart-beat messages from a worker to the master so that the master, which is on the critical path of all decisions, would not be overloaded. But a larger gap is more likely to leave idle machines unused for a longer period of time during the gap. Second, the significantly larger number of tasks simply exceeds the capability of a single scheduler. Specifically, our previous scheduler could only handle task submission rates in the range of a few thousand tasks per second, but the average task submission rate is 40K/sec (and the peak is 61K/sec) for the 30 days in Figure 1a.

Objectives and constraints. In addition to the scalability challenges posed by both the workload and the cluster size, the new scheduler should also achieve a good trade-off among various scheduling objectives. We focus on the following objectives: (1) *scheduling efficiency or delay*, i.e., how long a task waits for its required resources to be allocated; (2) *scheduling quality*, i.e., whether the resource preference of a task (e.g., machines where its data are stored, machines with larger memory or faster CPUs) is satisfied; (3) *fairness* [19] and

that is allocated with a certain amount of resources according to a project's available resource quotas.

priority; and (4) *resource utilization*. These objectives often contradict each other. For example, high scheduling quality may prolong the scheduling delay; maintaining strict fairness can leave resources unused and thus hurt utilization.

Over the years, the complicated logic for balancing these objectives has been programmed into various scheduling strategies. In addition, other cluster components such as application masters and worker agents are maintained by other teams in Alibaba, and it takes years' collaborative efforts to make them robust and behave as expected. Thus, *the new scheduler design should make as few changes to the existing codebase as possible, so as to ensure the entire system's robustness and backward compatibility (e.g., other cluster components need not be changed and the existing scheduling strategies can be applied in the same way)*. Finally, *the system upgrade should be transparent to both the internal users and cloud clients of Alibaba*.

Methodology. We first investigate existing works that may serve as a potential solution and pick the most suitable one for further analysis. We then develop a simplified model to gain insights mathematically. As it is hard to acquire an accurate mathematical solution and the simplified model also leaves out critical factors that cannot be dismissed in a real production environment, we use a simulation to find out the determining factors, which are used to derive our solution to the scalability problem. Finally, we evaluate our solution in a high-fidelity simulation cluster with workloads sampled from the production clusters as shown in Figure 1b.

3 Can We Adopt an Existing Scheduler?

As many schedulers have been proposed, we first examine if our problem can be addressed by an existing scheduler.

3.1 Shared-State Architecture

Omega [39] proposed the *shared-state scheduler architecture* as described in §1. This architecture addresses the limitations in *monolithic*, *two-level*, and *static partitioning* approaches, respectively. First, each scheduler in the shared-state architecture can run a different scheduling strategy programmed in separate code bases for different types of jobs, as to avoid the software engineering difficulties of maintaining all strategies in one code base and using multi-threading for solving head-of-line blocking in *monolithic* architecture. Second, each scheduler has a global view of the cluster, thus solving the problem of limited visibility in *two-level* schedulers such as Mesos [25]. This allows global policies (e.g., fairness and priority) to be implemented. Third, each scheduler can assign tasks to any machine in the cluster instead of a fixed subset of partitions, which reduces resource fragmentation in *statically partitioned* clusters and achieves higher utilization [45].

The shared-state architecture is a neat design that also satisfies our two hard constraints given in §1. First, interactions between application masters, worker agents and schedulers would require little change. An application master still talks to only one scheduler and performs application-level scheduling on a pool of committed resources. An agent reports the status of each task to its corresponding scheduler in the same way. Thus, backward compatibility is ensured. Second, users rely on the behaviors of global policies such as fairness and priority to organize their workflows to ensure job completion before deadline and effective resource utilization. They may have accumulated many scripts for arranging job submissions over the years. With global policies unchanged, these scripts can be reused.

With the hard constraints resolved, we move on to analyze whether this architecture also meets our other design goals given in §1, i.e., scalability, low latency and high scheduling quality. We present our analysis in §4, but before that, we also consider other alternatives below.

3.2 Schedulers Focusing on Scalability

Shared-state approaches. Apollo [9], Mercury [30] and Yaq-d [37] also use a shared state. They do not resolve conflicts by a centralized coordinator as in Omega, but let schedulers push tasks to distributed queues in workers. Based on task duration estimation, Apollo shares a *wait-time matrix (WTM)* as the state to keep the queue length in each worker. The WTM allows a scheduler to infer future resource availability instead of being limited to present resource availability. Yaq-d discusses queue sizing and reordering strategies based on wait time. In Mercury, tasks with guaranteed resources are handled by a centralized scheduler in order to enforce fairness and priority, while other tasks are handled by distributed schedulers.

These schedulers are not as general as Omega's design (e.g., their designs cannot be easily adopted in cluster) for the following reasons. First, task duration estimation can be inaccurate due to little prior information, changing input data size, data skewness [11, 33] and temporal interference [8, 16]. Task duration estimation is particularly difficult for modern clusters (e.g., clouds, our data centers) as they process diversified tasks from both internal and external users running on a broad range of systems. Second, predicted resource availability improves scheduling quality primarily due to disk locality, e.g., in Apollo's cluster [9]. A scheduler would assign a task to a busy machine as long as the benefit from disk locality is greater than the extra time the task waits to be executed. However, disk locality becomes less important [7] when faster switches, NICs [1, 6] and compressed file format [2–5, 42] are in use. Third, distinguishing tasks as in Mercury complicates the intra-job scheduling design of the applications, which compromises backward compatibility.

Other approaches. There are also distributed schedulers that do not share a cluster state. Sparrow [36] uses batch sampling and late binding to improve scheduling efficiency and task completion time. Tarcil [17] extends Sparrow by dynamically adjusting sampling sizes. Hawk [14] dispatches long batch jobs to a centralized scheduler, which adopts the least-waiting-time strategy, and short jobs to a set of distributed schedulers. Hawk also reserves a small portion of the cluster for short jobs and uses task stealing to avoid short tasks being blocked by long tasks. Eagle [13] takes one step further to proactively avoid assigning a short task to a machine with long tasks running or waiting. It also enables a worker machine to repetitively fetch remaining tasks from a job to mimic the least-remaining-time-first strategy.

The designs of these schedulers are also not general and adopting them has the following key concerns. First, they also rely on accurate task duration estimation as in shared-state approaches. Second, Sparrow does not have a global view of a cluster to implement global policies, and its strategy is customized for fine-grained tasks of Spark jobs [35] and the duration of such tasks is in the range of milliseconds. Hawk and Eagle focus on using a single scheduling algorithm to reduce the JCT of homogeneous workloads. In contrast, Omega’s design offers a global cluster view and allows multiple schedulers to run different scheduling algorithms.

3.3 Other Related Work

Schedulers such as YARN [44], Mesos [25], Borg [41,45] and Kubernetes [32] consider various issues in production clusters such as generality, extensibility, robustness and resource utilization, in contrast to schedulers discussed in §3.2 which focus on high scalability and low scheduling latency. Apart from scheduler architectures, many scheduling algorithms have been proposed. [23, 29, 31] focus on intra-job task scheduling to optimize job completion time. Job scheduling algorithms that aim to achieve objectives such as fairness [10, 19, 27, 47], high resource utilization [12, 15, 16, 20, 20, 21, 27, 28, 34, 40, 46], job completion time [22, 43], and workload autoscaling [38] may also be adopted in our schedulers according to the application needs of our clusters. These works do not focus on scheduler architecture design and are orthogonal to our work.

4 An Analysis on Scheduling Conflicts

In §3 we singled out Omega’s shared-state architecture [39] as a potential solution to our problem. As mentioned in §1, *multiple schedulers may contend for the same piece of resource*, which leads to **conflicts** and hence scheduling delay. Thus, we want to study the following questions: (1) *What are the factors that determine the conflict rate? how important is each of these factors?* (2) *How bad can the scheduling delay be?* (3) *How can we avoid or alleviate the scheduling delay? What price do we need to pay?*

4.1 Conflict Modeling

We first quantify conflicts by constructing an analytical model for scheduling using the shared-state architecture presented in §1. The scheduling of a task may be delayed when there are not enough available resources to be allocated for the submitted tasks, or when there are enough available resources but the scheduler cannot keep up with the task submission rate. As we focus on the scheduler design, we only investigate the second case, i.e., the scheduler is the potential bottleneck of allocating available resources to tasks in time.

One extreme case that puts schedulers on the test is when the task submission rate and the resource needs of the tasks just match with the total amount of resources in a cluster. Suppose that there are S slots of resources in the cluster and each task takes one slot for its execution. At each unit of time (UT), J tasks are submitted for scheduling and each of the tasks runs for a fixed duration of T UT. If we have a single ideal scheduler that assigns tasks without delay and incurs no conflict, then all the J tasks will be committed immediately as long as there are available slots. We say that a task is **committed** when its requested resources are allocated. In reality, however, we do not have such an ideal scheduler. Instead, we have multiple schedulers that can lead to conflicts. By queuing theory, the scheduling delay will grow to infinity. However, as we will discuss below, this extreme case helps us see how much overhead due to conflicts is added by introducing multiple schedulers and how much price we should pay to fix this problem. We are particularly interested in this extreme case as it gives the scheduler the most pressure.

Now suppose that a scheduler can only schedule and commit $K < J$ tasks within each UT *given that there is no conflict*. To keep up with the task submission rate, i.e., J tasks/UT, we need to use at least $N = J/K$ schedulers to share the scheduling load. We assume that tasks are uniformly distributed to the schedulers and each scheduler independently makes its own scheduling decision based on the latest synchronized local cluster state, where available slots are randomly chosen for assignment. We will discuss the implications of these assumptions in §4.2.

We name the above setting as the **Baseline**. As conflicts cannot be avoided in reality, a scheduler cannot commit all its K tasks within the current UT when conflicts occur, which leads to scheduling delay. We investigate how many conflicts can be incurred. Consider that the cluster has S_{idle} idle slots. We denote the number of schedulers picking slot i as a random variable X_i . Since each scheduler picks a slot with probability $\frac{K}{S_{idle}}$, X_i follows a binomial distribution. We denote the number of conflicts at slot i as a random variable $Y_i = \max(X_i - 1, 0)$.

Then, we can deduce the expectation of Y_i as:

$$\begin{aligned}
\mathbb{E}(Y_i) &= 0 \cdot \Pr\{X_i \leq 1\} + \sum_{j=1}^{N-1} j \cdot \Pr\{X_i = j + 1\} \\
&= \sum_{j=1}^{N-1} (j+1) \cdot \Pr\{X_i = j+1\} - \sum_{j=1}^{N-1} \Pr\{X_i = j+1\} \\
&= \mathbb{E}(X_i) - \Pr\{X_i = 1\} - (1 - \Pr\{X_i = 1\} - \Pr\{X_i = 0\}) \\
&= \frac{NK}{S_{idle}} - 1 + \left(1 - \frac{K}{S_{idle}}\right)^N \quad (1)
\end{aligned}$$

Since we have S_{idle} idle slots in total, the expectation of the total number of conflicts is given by:

$$\mathbb{E}\left(\sum_{i=1}^S Y_i\right) = S_{idle} * \mathbb{E}(Y_i) = NK - S_{idle} + S_{idle} * \left(1 - \frac{K}{S_{idle}}\right)^N \quad (2)$$

To reduce conflicts, naturally we may consider to add more slots (so as to reduce the contention for resources) or more schedulers (so as to increase the capacity to handle conflicts). We analyze each of them as follows.

Adding extra slots. Merely adding extra slots to a cluster can never reduce the expectation of the number of conflicts to 0 according to Eq.(2). We may reduce conflicts by adding more extra slots, but Eq.(2) shows that the effect of increasing the number of extra slots (i.e., S_{idle}) is superlinearly diminishing.

Adding extra schedulers. In the Baseline setting, each scheduler is operating at its full capacity and does not have room to resolve conflicts. By adding more schedulers, each scheduler may handle less than K tasks in each UT and now have time to reschedule tasks due to conflicts. However, this may lead to more conflicts, as the following analysis shows. Suppose now we have $A * N$ (where $A > 1$) instead of N schedulers, and thus each scheduler has $\frac{K}{A}$ tasks to schedule. Substituting N with $A * N$ and K with $\frac{K}{A}$ in Eq.(2), we obtain:

$$f(A) = \mathbb{E}\left(\sum_{i=1}^S Y_i\right) = AN \frac{K}{A} - S_{idle} + S_{idle} * \left(1 - \frac{K}{AS_{idle}}\right)^{AN} \quad (3)$$

$$= C_1 + C_2 * \left(1 - \frac{1}{x}\right)^{x * C_3}, \quad (4)$$

where $C_1 = NK - S_{idle}$, $C_2 = S_{idle}$, $C_3 = \frac{NK}{S_{idle}}$, and $x = \frac{AS_{idle}}{K}$.

In Eq.(4), since C_1 , C_2 and C_3 are independent of A and $C_2, C_3 > 0$, $f(A)$ increases if $\left(1 - \frac{1}{x}\right)^x$ increases. Since $\left(1 - \frac{1}{x}\right)^x$ increases monotonically as A increases when $x \geq 1$, $f(A)$ also increases as A increases, i.e., the expected number of conflicts increases when more schedulers are added. Thus, we need to find out *whether the overhead of having more conflicts can be covered up by the benefit of having more time for rescheduling tasks such that these tasks will be committed within the same UT in which they are submitted.*

Adding extra slots and schedulers. Due to the diminishing returns by adding extra slots only or schedulers only, it is reasonable to believe that adding both of them can lead to

a more cost-effective solution. However, as S_{extra} and A are intertwined in Eq.(4) and Eq.(4) only indicates the number of conflicts in a single round of scheduling, it is hard to derive an accurate mathematical solution for S_{extra} and A to achieve 0 scheduling delay in a series of rounds of scheduling. Nevertheless, we can make use of the equations we have derived for quantifying conflicts to construct a simulator for the scheduling process. By simulating different combinations of N, J, K, S , with the constraints $J * T = S$ and $N * K = J$, we can examine the minimal requirements for S_{extra} and A .

4.2 The Implications of Our Assumptions

Before we present the simulation, we remark that although the assumptions made in §4.1 lead to an easier analysis, some of them diverge from the reality in the following aspects:

[A1] It is assumed that schedulers pick slots for tasks in a uniformly random fashion. In reality, some machines may be preferred (e.g., because of more advanced hardware or data locality) and result in more conflicts than the expectation.

[A2] It is (implicitly) assumed that all the schedulers schedule tasks synchronously round after round, and the local cluster states can be synchronized with the master state as frequently as we want. However, in reality each scheduler acts asynchronously with each other. We also cannot piggyback the synchronization of the cluster state on the return trip of every commit request as in Omega [39], as explained in Footnote 1 in §1. Instead, a time gap G is set between two synchronizations of a local state with the master. In-between the gap, when a scheduler commits a task to some slot, the slot's status in the master state and the scheduler's local state is updated. Such updates make the local states of other schedulers *stale*, and scheduling decisions based on a stale view of the state lead to more conflicts than indicated by our equations in §4.1.

[A3] We assume conflicts are uniformly incurred on schedulers. But in reality commit requests from schedulers are handled by the master in an FIFO manner and thus schedulers whose requests are sent earlier will get fewer conflicts.

4.3 Simulation Analysis

We construct a simulator following most of the setting in §4.1. We also integrate the factors listed in §4.2 into the simulator as follows.

To address [A1], we assign a score to each slot based on a normal distribution and schedulers pick desired slots by weighted sampling according to their scores (instead of in a uniformly random fashion as in §4.1). We vary the variance of slot scores to control how much slots should be contended.

To address [A2], we set a **synchronization gap** G . At each time period G , schedulers synchronize their local state with

the master state. Each scheduler makes its scheduling decisions independently based on its own local state, which may become stale until its next synchronization. Since G controls the staleness of the local states, we vary G to examine its impact on scheduling delay.

To address [A3], commit requests are sent to the master asynchronously by schedulers in our simulation and the requests are processed in FIFO. We also distribute the master state by partitioning so that commit requests to a slot are only sent to the partition that contains the slot. We vary the number of partitions to examine the effects of having a long queue at a single master versus shorter queues at distributed partitions.

Simulation setting. We assume that the cluster has 200K slots, and each task takes 1 slot and uses it for 5s, which is to simulate a typical scenario in our cluster, i.e., task submission rate is around 40K tasks/s and the duration of most tasks is between 1s to 10s (§2). Suppose that a scheduler takes 0.25ms to schedule a single task (close to our real scheduling latency) and thus we need at least 10 schedulers to just match with the 40K/s task submission rate (assuming no conflicts). Tasks are submitted in batches, where each batch has 100 tasks, and 10 batches are submitted to each scheduler in each second. Each scheduler makes scheduling decisions for a whole batch and schedules the next batch only after the tasks in the current batch have all been committed.

We vary task submission rate R , synchronization gap G , slot scores, and the number of partitions P of the master state, to examine how S_{extra} changes with $A = 2, 4, 8$. We want to find which factors are the main contributors to conflicts. For each simulation, we use the following *default setting*: $R = 40K/s$, $G = 0.5s$, a normal distribution of slot scores with mean = 10 and variance $V = 2$, and $P = 1$ (i.e., no partitioning).

In the simulation, we try to find the minimum number of combinations of A and S_{extra} to achieve a small scheduling delay. With a larger A , the number of tasks B in one batch is adjusted to $B_{new} = \frac{B}{A}$. With a larger S_{extra} , the number of conflicts decreases. In the simulation, tasks are submitted at a fixed rate for 90 seconds (similar patterns are observed after 90s), and we say that the scheduling delay is small when the simulation finishes in less than $110\% \times 90s = 99s$.

Simulation results. Figures 2a, 2b and 2c show that a significant number of extra slots needs to be added as we increase R , G or V . This is because a larger R means more work for each scheduler, a larger G means making scheduling decisions based on a staler state and for longer time, and a larger V means more contentions for higher-quality slots, all of which lead to more conflicts. To maintain the same scheduling delay, more extra slots can help reduce the contentions for slots and hence conflicts. However, Figure 2d shows that the impact of P is minimal. This is because even when the master state is not partitioned, some schedulers may have more conflicts at one point of time but the number of conflicts evens out among all schedulers over time.

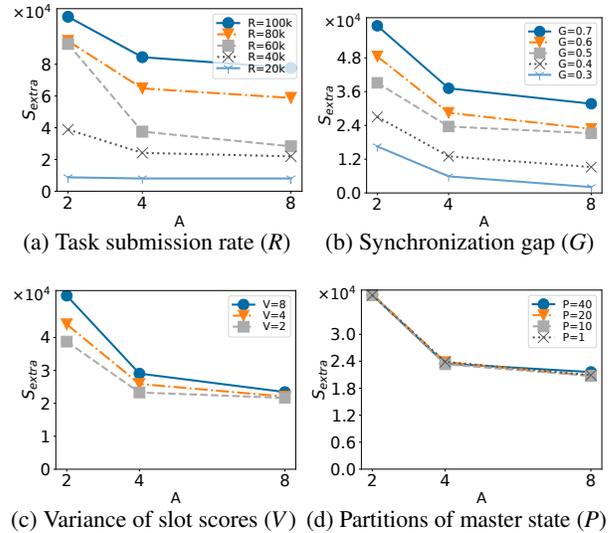


Figure 2: Simulation results

In each simulation, we also increase the number of schedulers to 2, 4 and 8 times more to see how a larger A helps reduce scheduling delay. Figures 2a-2d consistently show that increasing A has diminishing benefits. This conforms with our findings in §4.1 as sharing the same total amount of work by more schedulers leads to more conflicts. Moreover, it is costly to operate many schedulers in a real cluster, as we use active standbys to replace failed ones to meet our SLAs.

Adding a large number of extra slots and schedulers is a big price to pay for a large-scale production cluster, considering both the extra machine costs and daily operation costs. Thus, we want to look for a new solution. According to the simulation results, task submission rate, slot scores and synchronization gap are the main factors that contribute to conflicts. However, we cannot change task submission rate as it is fixed by the workloads. Slot scores are related to the resources desired by tasks, which may be compromised (e.g., a task may also accept a fast machine instead of the fastest machine) to alleviate the contentions and hence reduce conflicts. Synchronization gap affects performance as it controls the staleness of the local states, which in turn determine the probability of conflicts. Our solution will focus on the last two factors, i.e., **slot quality** and **staleness of the local states**.

5 Partitioned Synchronization

As our solution aims to reduce conflicts, we first describe an alternative that offers no-conflict guarantee, namely *pessimistic scheduling*. One implementation of pessimistic scheduling is to let each scheduler have *exclusive partitions* of a cluster, but this can lead to resource under-utilization as some partitions may hold idle resources that can be used by other schedulers [39, 45]. Another implementation is by applying

locks on resource slots, but this can block the actions of other schedulers and increase scheduling delay due to reduced concurrent processing [24, 39]. We want to avoid the limitations of pessimistic scheduling and reduce conflicts in optimistic scheduling (as in Omega), while enjoying their benefits, i.e., *high concurrent processing while having few conflicts*.

5.1 The Design

Observation. We found that scheduling delay increases disproportionately within the period G . When the cluster state is just synchronized, it is fresher and scheduling has fewer conflicts. But when the state becomes more outdated towards the end of G , scheduling decisions result in more conflicts. Conflicts lead to rescheduling, which may in turn cause new conflicts, and hence rescheduling recursively. Consider our default simulation setting in §4.3 and $S_{extra} = 15,000$, and divide G into two equal intervals: 0s-0.25s and 0.25s-0.5s. The average conflict rate in the first interval is 48% and that in the second interval is increased to 64%. As a result, the average delay of the first interval is only 23% of the total average delay, while the second interval contributes to 76% of the total. In other words, most of the delay is caused by the staler view of the cluster state in the later interval of G .

Main idea. Our main idea is *to reduce the staleness of the local states and to find a good balance between resource quality (i.e., slot score) and scheduling efficiency*, as these two are the main contributors to conflicts according to our findings in §4.3. We present our solution, **partitioned synchronization (ParSync)**, as follows.

ParSync logically partitions the master state into P parts. Assume that we have N schedulers, where $N \leq P$. Each scheduler still keeps a local cluster state, but *different schedulers synchronize different partitions of the state each time*, in contrast to *synchronizing the entire state* (let us denote it as **StateSync**) as in the approach discussed in §4. Specifically, assume (for simplicity) P is a multiple of N , the i -th scheduler starts its synchronization from the $(\frac{P}{N} * (i - 1) + 1)$ -th partition to the $(\frac{P}{N} * i)$ -th partition of the state in each round of synchronization, and the subsequent rounds of synchronization continue in a round-robin manner. Thus, in each round, all schedulers synchronize different partitions of the cluster state. This is also why we require $P \geq N$, as otherwise some schedulers would synchronize the same partition(s).

How does ParSync work? First, each scheduler has a *fresh* view of $\frac{P}{N}$ partitions of the cluster resources, so that the scheduler can commit its tasks to available slots in these partitions with a high success rate (since its view on these partitions is fresher than that of any other schedulers). This design is particularly favorable for scheduling short tasks in a highly contended cluster like ours. For short tasks, it is more critical to acquire resources sooner for their execution, instead of spending long time to find the most suitable slots because bet-

ter slots may not compensate for the scheduling delay (which itself can be comparable with or even longer than the execution time of a short task). Note that the majority of the tasks in our workloads are short tasks, as shown in Figure 1b.

Second, ParSync also effectively reduces the average *staleness* of the entries in a local cluster state, which we explain as follows. As the granularity of synchronization is changed from the entire cluster state to $\frac{P}{N}$ partitions, which effectively changes the synchronization gap to $\frac{G}{N}$. In contrast to synchronizing the entire state at every G time units, with ParSync each scheduler synchronizes $\frac{P}{N}$ partitions of its local state at every $\frac{G}{N}$ time units. Now let us define the **staleness** of a partition as the period of time since its latest synchronization, and let **AS** be the **average staleness** of all the partitions of a local state. Whether ParSync is used or not, the average of “AS”s over all time points is always $\frac{G}{2}$. However, ParSync reduces the variance of “AS”s, which we explain as follows.

Suppose that there is only one partition, then the minimum and maximum AS are 0 and G (at time 0 and G , respectively). If there are two partitions, then the minimum and maximum AS are $(0 + \frac{G}{2})/2 = \frac{G}{4}$ and $(\frac{G}{2} + G)/2 = \frac{3G}{4}$. Thus, when we increase the number of partitions from 1 to 2, we also bound the AS in the range of $[\frac{G}{4}, \frac{3G}{4}]$ instead of $[0, G]$. Intuitively speaking, using more partitions here is like sacrificing the period when the AS is small (i.e., $[0, \frac{G}{4}]$) to avoid the period when the AS is large (i.e., $[\frac{3G}{4}, G]$). But *this sacrifice can lead to significant reduction in the scheduling delay, because conflicts increase much faster in the later interval of G as we discussed in the Observation earlier*.

As having more partitions bounds the AS in a smaller range, consequently the variance of AS is also reduced. When the number of partitions becomes sufficiently large, we push the minimum and maximum AS close to $\frac{G}{2}$. Thus, *the eventual effect of ParSync on scheduling delay can be approximately viewed as reducing G to $\frac{G}{2}$, which also means that we reduce the average staleness of a local state*.

The above analysis is important because our Observation earlier also indicates that a reduction in the staleness of the local state can significantly reduce the number of conflicts and hence also the scheduling delay.

ParSync allows us to better balance resource quality and scheduling efficiency. For example, as conflicts are less likely to happen in periods when the cluster is not in intensive use, a scheduler may take higher risk of rescheduling and try to commit tasks to high-quality slots in other staler partitions (instead of its freshest $\frac{P}{N}$ partitions). In this case, ParSync may adopt optimistic scheduling to prioritize resource quality (i.e., slot score). In contrast, a scheduling strategy aiming for successful commits may take a pessimistic approach (but with all schedulers concurrently working). In addition, each scheduler still has a global view of the cluster and hence global scheduling policies can also be implemented.

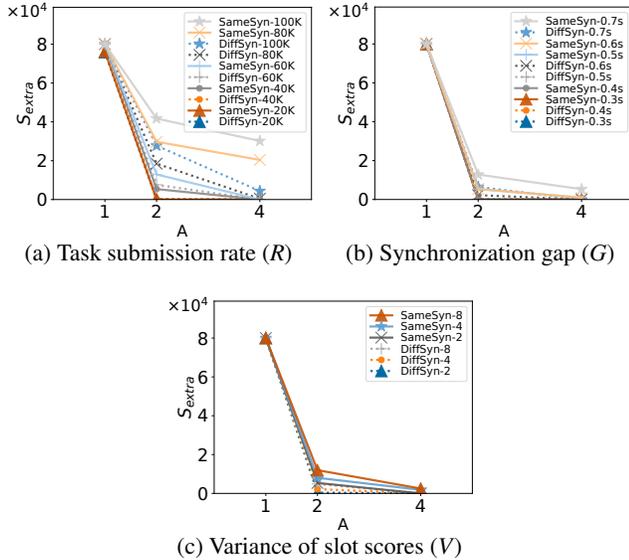


Figure 3: The effects of different synchronization strategies

5.2 Simulation Analysis on ParSync

We further analyze the effectiveness of ParSync by simulation. In all the simulations, we use the same setting and the default values of the parameters as in §4.3.

Synchronization strategies. We first compare two synchronization strategies: (1) **DiffSyn**: each scheduler synchronizes different partitions of the cluster state, i.e., the strategy used in ParSync; (2) **SameSyn**: all schedulers synchronize the same partitions, i.e., all schedulers synchronize the $(\frac{P}{N} * (i - 1) + 1)$ -th to $(\frac{P}{N} * i)$ -th partitions in the i -th round, for $1 \leq i \leq N$. In this simulation, a scheduler may pick slots from any partition in the cluster (i.e., using optimistic scheduling).

We vary task submission rate R , synchronization gap G , the variance of slot scores V , and the number of schedulers (i.e., AN) from N to $4N$, where $N = 10$. As shown in Figures 3(a)-(c), SameSyn requires more extra slots than DiffSyn in order to achieve the same latency, which can be explained as follows.

We found that schedulers tend to pick slots from fresher partitions. This is because when a scheduler tries to commit a task to a slot, it also updates the status of the slot in its local partition to “taken” (either by this task or already taken). Over time, more and more slots in the staler partitions are marked “taken”, which are refreshed until the partitions are synchronized. This becomes a problem for SameSyn because all schedulers are competing for available slots from the same fresher partitions, thus leading to more conflicts. In contrast, under DiffSyn, the fresher partitions of each scheduler are different, meaning that each scheduler always has higher commit rate in its own fresher partitions. Each scheduler under DiffSyn still has high conflict rate when committing tasks to its

staler partitions, but this situation also happens to SameSyn.

Scheduling strategies. We examine three scheduling strategies: **Quality-first**, **Latency-first**, and **Adaptive**. In Quality-first, a scheduler schedules a task by first choosing the partition with the highest average slot score and then picking available slots by weighted sampling based on slot scores. In Latency-first, each scheduler schedules a task by first picking slots from its freshest partitions, but if suitable slots are not found, then it looks for slots in other partitions (from the least stale partition first).

Adaptive uses Quality-first when it does not incur much scheduling delay, while it adopts Latency-first when scheduling efficiency is more critical. It calculates an exponential moving average (EMA) of scheduling delay, such that Quality-first is used when the EMA is smaller than a threshold τ , and Latency-first is used otherwise. Note that τ is usually the SLA for the scheduling delay specified by the cluster operator or users. Here we intend to show that the adaptive strategy can bound the scheduling delay by τ when the cluster is busy, and attain high slot quality as Quality-first when the cluster is not busy, instead of arguing what is the best trade-off we can make. We run simulation by setting $\tau = 1.5s$ as a demonstration.

We simulate three scenarios that may happen in our production cluster on a typical day as follows. We create two groups of schedulers, A and B, and divide the timeline into three phases: (1) A and B are both operating at 2/3 of their full capacity; (2) A is operating at full capacity while B remains the same; (3) A and B are both operating at full capacity. Each phase is run for 30s.

Figure 4 plots the median and the 10th-90th interval of the slot scores and scheduling delay of Group A (dark color) and Group B (pale color), respectively. We also plot the cluster utilization rate as a thick dashed curve in Figures 4b, 4d and 4f for reference.

During Phase 1, the three strategies have similar slot quality and scheduling delay. This is because the task submission rate is only 2/3 of the full load and the cluster is only 60%-70% utilized, and thus all schedulers (from both groups) have high-quality slots to pick without incurring many conflicts.

During Phase 2, the three strategies start to behave differently as Group A is now undergoing a stress test. When Quality-first is used, Figure 4a shows that the slot quality is not degraded compared with Phase 1; but Figure 4b reveals the real problem as the scheduling delay of Group A schedulers surges, while that of Group B remains stable. Thus, Quality-first is not effective when the scheduling load is high. When Latency-first is used, Figures 4c&4d show that both the slot quality and scheduling delay of both groups only become slightly worse. This is because even though Group A is fully loaded, the overall cluster utilization rate is only about 80% and thus the schedulers can still have enough quality slots to pick from their freshest partitions. When Adaptive is used, Figure 4f shows that the scheduling delay of Group A

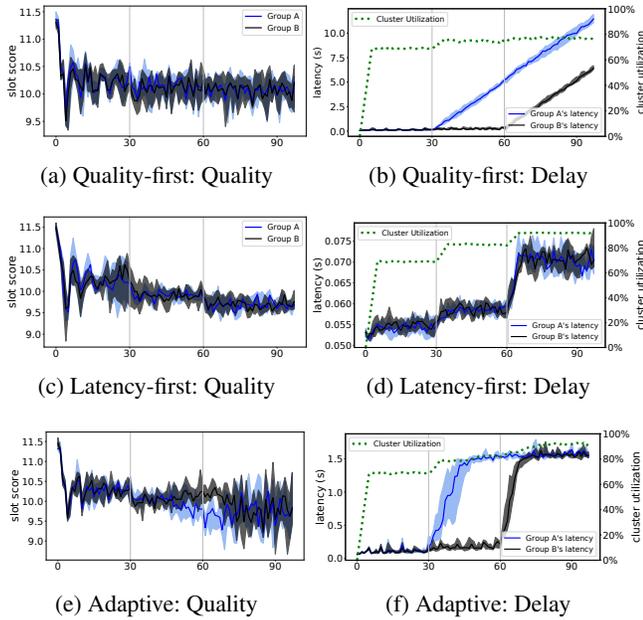


Figure 4: The performance of different scheduling strategies (dotted green line denotes the cluster utilization)

increases to 1.5s and then remains bounded there, as τ is set at 1.5s (note that this is much lower than the delay of Group A in Figure 4b, which rises to 4.7s). This means that Group A schedulers now switch to use Latency-first, while Group B still use Quality-first. As a result, Group A now has lower slot quality than Group B as shown in Figure 4e. Although Group B uses Quality-first while Group A uses Latency-first, the scheduling delay of Group B is actually much lower than that of Group A in Phase 2 of Figure 4f because Group B is only operating at 2/3 of its full capacity.

During Phase 3, even though Quality-first maintains the same slot quality, the scheduling delay of both groups keeps increasing. The slot quality of both Latency-first and Adaptive worsens compared with Phase 2 due to the heavier load and now the schedulers are competing more for quality slots. Figure 4d shows that the delay of Latency-first for both groups also increases, though still much lower than Quality-first and Adaptive. Adaptive bounds the delay of both groups at 1.5s.

In summary, Quality-first works well when the scheduling load or the cluster load is not heavy, as it leads to high-quality slot allocation. When the load is heavy, the delay of Quality-first may become too high and Latency-first is preferred as it achieves both high efficiency and slot quality. Adaptive is also a good choice as Adaptive keeps the latency bounded (which is particularly desirable as the bound can be tied to the SLA), while Adaptive can also take advantage of Quality-first to get high-quality slots when the load is not heavy.

To deploy ParSync in our cluster, we discuss some details to important issues such as what changes are needed in the upgrade, how the scheduling objectives are achieved, and how

the cluster is partitioned in Supplemental Material B in [18].

6 Performance Evaluation

We conducted our experiments in a high-fidelity testing environment called *Wind Tunnel*. Note that although ParSync is deployed in large production clusters in our company, conducting real experiments to evaluate ParSync’s scalability on these large clusters is prohibited by company policy as not to cause unexpected interruption to normal business operations. The company relies heavily on *Wind Tunnel* to test the robustness and performance of every cluster scheduler component and scheduling algorithm, and any change must be tested extensively in *Wind Tunnel* before applied to the production clusters.

The results obtained from *Wind Tunnel* are close to the true performance of the schedulers in our production clusters as *Wind Tunnel* uses the same codebase of the entire production cluster system and it incorporates critical factors from our production environment. First, in *Wind Tunnel*, each scheduler or resource manager is deployed on an independent machine and executes code used in production. A set of machines is then used to simulate resource slots in a production cluster. The interaction between schedulers, resource managers, and workers are the same as in production. *Wind Tunnel* makes the staleness of local states to be more reflective of real situations by deploying resource managers and schedulers on a real network topology. Second, the schedulers execute the same scheduling logic as in our production cluster, which enables the evaluation to reproduce the true scheduling capacity of each scheduler. Third, each job has a list of *preferred slots* computed at runtime for its tasks (instead of using a static score for each slot as in the simulations in §5.2) to better reflect the contentions on slots in reality.

The main difference between the high-fidelity simulation by *Wind Tunnel* and a production cluster is that the execution of a task is effectively “sleeping”, and each worker machine in *Wind Tunnel* is simulating many worker machines in a production cluster. All the rest are based on the same code base. Therefore, the results obtained from *Wind Tunnel* are similar to the results obtained from the production clusters.

Settings. In the experiments, we used the following default settings unless otherwise specified. We deployed 20 machines in *Wind Tunnel* for schedulers and 2 machines for resources managers. We used another 30 machines in *Wind Tunnel* to simulate 200k slots and each resource manager oversees 100k slots. The workload we used was sampled from the production trace whose statistics is shown in Figure 1. The baseline peak task submission rate from the sampled workload is around 40k/s, and we varied the rate to 50%, 80% and 95% of 40k/s to simulate different levels of pressure. The cluster state was partitioned into 20 parts in ParSync. The synchronization gap in our production cluster is around 0.5s.

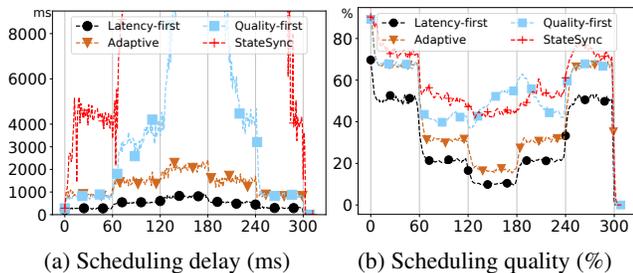


Figure 5: Performance of ParSync and StateSync

Performance metrics. We use *scheduling delay* and *scheduling quality* as the major measures. Scheduling quality is the percentage of tasks that are allocated to their preferred slots. We also report the *scheduling throughput* and the *number of conflicts* to analyze the performance.

6.1 Performance Comparison

There are many works related to cluster scheduling (§3). The scheduling algorithms mentioned in 3.3 are orthogonal to our work as many of them can also be applied for scheduling in ParSync. As discussed in §3.2, most existing schedulers are not as general as Omega’s shared-state design (§3.1), as they have specific requirements (e.g., accurate task duration estimation, strong disk locality) or cannot provide important features such as backward compatibility, support of multiple scheduling algorithms and global view. While these schedulers are effective in their own settings, we do not compare ParSync with them since the design objectives are different (and it is also difficult to create an environment in Wind Tunnel for a fair comparison, if possible, with these systems).

Instead, we compared ParSync with StateSync, which simulates Omega. We also tested StateSync with only 1 scheduler, which simulates our previous centralized scheduler (similar to YARN), but the delay is nearly two orders of magnitude worse than StateSync and thus we do not report the details.

For ParSync, we tested Latency-first, Quality-first and Adaptive ($\tau=2s$) introduced in §5.2. StateSync adopts the shared-state architecture in §4, which synchronizes the entire state each time. Both ParSync and StateSync used 20 schedulers. Tasks were submitted during a 300-second period, which was divided into five 60-second phases with different submission rates: 50%, 80%, 95%, 80%, 50% of 40k/s. The 50%, 80% and 95% rates simulate medium, medium-heavy and heavy loads of a cluster.

Figure 5a shows that the average scheduling delay of ParSync using Latency-first is significantly smaller than the other approaches throughout the five phases. The delay of ParSync using Adaptive is bounded by τ . The delay of Quality-first increases much faster than Latency-first and Adaptive dur-

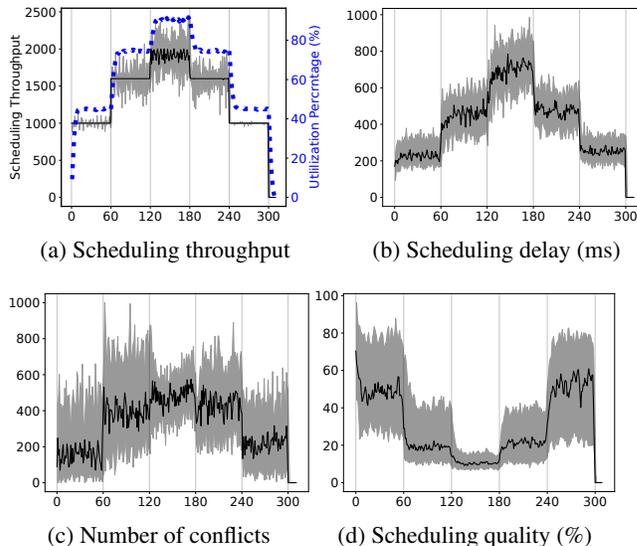


Figure 6: Results of ParSync using Latency-first

ing the peak periods (e.g., Phase 3), but is still much smaller than that of StateSync. Starting from Phase 2, StateSync’s delay increases rapidly as its schedulers are not able to handle the high task submission rate and uncommitted tasks starts to accumulate. The high delay starts to drop only in Phase 5 when the accumulated tasks from the previous phases are gradually committed. Although Figure 5b shows that the *scheduling quality of StateSync is actually quite good, its high scheduling delay makes it infeasible for production use.*

The results verify our findings in §4 that *sharing the whole cluster state without partitioned synchronization cannot scale to handle high task submission rates in our cluster, as the number of conflicts and the scheduling delay increase rapidly.*

6.2 Scheduling Delay and Scheduling Quality

Next we analyze in details the performance of ParSync, as reported in Figure 6 and Figure 7. In each figure, the dark curve plots the median and the shadows plot the 10- and 90-percentile values among the 20 schedulers.

Scheduling throughput. Figures 6a and 7a show that both Latency-first and Quality-first can handle the task submission rate. In Phase 3, schedulers using Quality-first have throughput lower than 1.9K/s in Figure 7a (note that on average, each of the 20 schedulers receives 1K, 1.6K, 1.9K, 1.6K, and 1K tasks per second in each of the five phases). This is because of the surge in the scheduling delay in Phase 3 (Figure 7b), and hence some tasks are accumulated and only successfully committed during Phase 4. This also explains why the throughput of most schedulers goes above 1.6K/s at the beginning of Phase 4. Figures 6a and 7a also show that the

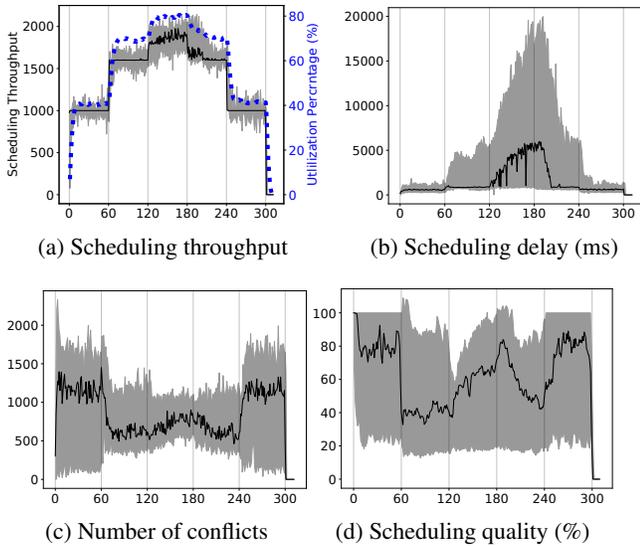


Figure 7: Results of ParSync using Quality-first

cluster utilization (plotted in thick dash curves) is reflected by the scheduling throughput in each phase.

Scheduling delay. Figure 6b shows that the scheduling delay of Latency-first increases with the task submission rate, which is mainly due to increasing scheduling conflicts as shown in Figure 6c. The median and mean delay roughly follow the task submission rate in each phase. We also observed that the total number of uncommitted tasks accumulated by each scheduler in each second is small and does not keep growing even during Phase 3 (see detailed results in Supplemental Material A.1 in [18]). This demonstrates that Latency-first is able to sustain and provide reasonable latency under heavy scheduling loads even if the peak period continues. In contrast, Figure 7b shows that using Quality-first, the scheduling delay increases rapidly in Phase 2 and Phase 3. The delay starts to drop in Phase 4 but is still high because it needs to clear the accumulated uncommitted tasks, which surges from 1,700 in Phase 2 to 5,800 in Phase 3 (1,700 and 5,800 are median values, details are reported in Supplemental Material A.1 in [18]). This is because Quality-first has a higher scheduling overhead as it checks all partitions to find preferred slots, its delay increases quickly when there are a large number of tasks to schedule. We also remark that Quality-first’s high delay is not primarily due to conflicts, as the number of conflicts of Quality-first is not much larger than that of Latency-first in Phase 3 as shown in Figures 6b and 7c.

Number of Conflicts. Figure 6c shows that Latency-first has an insignificant number of conflicts in Phase 1 as there are plenty of idle slots inside the freshest partition of each scheduler. There are more conflicts during Phase 2 to Phase 4 as

a scheduler may schedule some tasks to other less fresh partitions due to insufficient idle slots in its freshest partition. Interestingly, we observe an inverse pattern in Figure 7c, as Quality-first has a higher number of conflicts in Phase 1 and Phase 5 than in other phases. We examined the details and found that, in Phase 1 and Phase 5, Quality-first schedulers schedule tasks to preferred slots in all the partitions evenly since they all have plenty of idle slots. But from Phase 2, each scheduler starts to have an *increasingly* more idle slots in its own fresher partitions than in staler partitions (as discussed in §5.1). Thus, each scheduler tends to schedule more tasks to its fresher partitions, which incurs fewer conflicts.

Scheduling quality. Figures 6d and 7d show that Quality-first achieves considerably better quality than Latency-first. Note that there are always some tasks that may not get their preferred slots, as tasks are competing for preferred slots that may not be enough for all the tasks wanting them. Thus, the percentage of tasks getting their preferred slots drops accordingly when more tasks are submitted in Phases 2-4. However, Quality-first is actually able to allocate most slots to tasks that prefer them, although this is at the cost of higher delay when the scheduling load is heavy.

Compared with the more ideal simulation results in Figure 4, we observe more variations in the results shown in Figures 6 and 7 due to the introduction of real factors in our production cluster. But overall, the results are still consistent with our findings in §5.2, except that the trade-off between latency and quality becomes more obvious in Figures 6 and 7. The results in Figure 5 also validate that StateSync cannot handle our high task submission rates as ParSync does.

6.3 The Performance of the Adaptive Strategy

In this set of experiments, we evaluated how the latency threshold τ affects the performance of ParSync using the Adaptive strategy, by setting $\tau = 1000, 2000$ and 3000 . The range $1000 \leq \tau \leq 3000$ is representative of our workloads and it is not common to set the delay bound higher than 3,000 ms. We ran the same five phases as in the experiments in §6.2.

Figure 8 reports the median values (more details are reported in Supplemental Material A.2 in [18]) of the scheduling delay and quality of all schedulers. For all values of τ , Adaptive’s performance in Phases 1 and 5 is similar to that of Quality-first (see Figure 7), because Adaptive uses Quality-first when the scheduling load is not heavy. From Phase 2 to Phase 4, while Quality-first’s delay increases quickly as reported in Figure 7b, Figure 8a shows that Adaptive can effectively bound the delay at τ . Whenever the EMA is greater than τ during Phases 2-4, Adaptive switches to use Latency-first and thus it does not accumulate uncommitted tasks as in Quality-first. Consequently, its delay also drops quickly from Phase 3 to Phase 4, in contrast to Figure 7b. However,

Table 1: The effects of slot quality: in each cell, *left middle right* values = *Latency-first Adaptive*($\tau=1000$) *Quality-first* results (**STET**: average sum of task execution time; **JSD**: average job scheduling delay; **JCT**: average job completion time)

Metric(sec)/ α	0.9			0.8			0.7			0.6			0.5			0.4		
STET	922	914	904	888	873	854	858	822	772	836	788	712	797	726	622	762	600	542
JSD	0.4	0.8	3.7	0.4	0.8	3.3	0.4	0.8	2.9	0.4	0.8	2.0	0.4	0.8	1.9	0.4	0.7	1.5
JCT	10.3	10.6	12.7	10.3	10.5	12.1	10.2	10.3	11.2	10.2	10.0	9.9	10.1	9.5	9.1	10.1	8.6	8.4

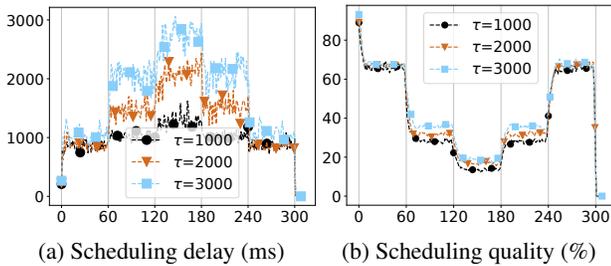


Figure 8: Results of the Adaptive Strategy

Figure 8b shows that Adaptive’s scheduling quality also drops in Phases 2-4, although its quality is still higher than that of Latency-first in Figure 6d because Adaptive uses Quality-first to get more quality slots until the EMA reaches τ . This is also why the quality of $\tau = 3000$ is generally better than those of $\tau = 2000$ and $\tau = 1000$.

Overall, *the results demonstrate the effectiveness of Adaptive in balancing scheduling delay and quality under different values of τ* . We report more details in Supplemental Material A.2, which show that we can choose different values of τ to bound scheduling delay without significantly affecting scheduling quality, throughput and the number of conflicts.

6.4 The Effects of Slot Quality

Running a task in its preferred slot reduces the task execution time (TET). The shorter the TET, the more room it creates for tolerating scheduling delay. Thus, if scheduling tasks to their preferred slots can shorten their TET significantly, Quality-first could be preferred to Latency-first. To test the effects of slot quality, we use a factor α to adjust the benefit of running a task in its preferred slot as follows: if the TET of running a task in a non-preferred slot is t , then its TET in a preferred slot is αt , where $0 < \alpha < 1$.

Table 1 reports the average sum of TET (**STET**, i.e., the sum of the TET of all tasks in a job, averaged over all jobs), average job scheduling delay (**JSD**, i.e., the duration between the time when a job is submitted and when all its tasks are committed), and average job completion time (**JCT**, i.e., the duration between the time when a job is submitted and when all its tasks are completed).

When we increase the benefit of preferred slots (i.e., smaller α), Quality-first is favored in terms of STET since it schedules more tasks to their preferred slots than both Latency-first and Adaptive as reported in §6.2 and §6.3. The JSD of Quality-first also decreases as α becomes smaller, because shorter TET releases occupied slots to other tasks earlier and hence leads to fewer conflicts. However, due to the higher scheduling overhead of Quality-first, its JSD is still much larger than that of Latency-first. In terms of JCT, since it benefits from both shorter TET and JSD, Quality-first starts to achieve a smaller JCT than Latency-first when the benefit of preferred slots exceeds its scheduling overhead, i.e., when $\alpha \leq 0.6$. In comparison, *the JCT of Adaptive always closely follows the best one of Quality-first and Latency-first, which proves the effectiveness of the adaptive strategy*.

6.5 Other Experimental Results

We also examined the effects of the number of partitions (which also changes the synchronization gap) and the scalability of ParSync. Due to the page limit, we report the details in Supplemental Material A.3 and A.4 in [18], and summarize the results here: (1) increasing the partition number has almost no impact on the scheduling performance; and (2) ParSync achieves stable performance as the scale of the cluster and task submission rate increases by 1 to 4 times of the capacity of our current cluster.

7 Conclusions

We presented ParSync, which increases the scheduling capacity of our production cluster from a few thousand tasks per second on thousands of machines to 40K tasks/sec on 100K machines. ParSync effectively reduces conflicts in contending resources to achieve low scheduling delay and high scheduling quality. The simplicity of ParSync allows us to maintain user transparency and backward compatibility that are essential to our production clusters.

Acknowledgments. We thank the reviewers for their constructive comments that have helped greatly improve the quality of the paper. This work was supported by the Alibaba-CUHK collaboration project “Large Scale Cluster Scheduling” (code: 7010574).

References

- [1] 40 and 100 Gigabit Ethernet. <http://www.extremenetworks.com>.
- [2] Apache Arrow - Powering Columnar In-Memory Analytics. <https://arrow.apache.org/>.
- [3] Apache Avro. <https://avro.apache.org/>.
- [4] Apache ORC - High-Performance Columnar Storage for Hadoop. <https://orc.apache.org/>.
- [5] Apache Parquet. <https://parquet.apache.org/>.
- [6] Technologies for Data-Intensive Computing. <http://www.hpts.ws/agenda.html>.
- [7] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. In *HotOS*, volume 13, pages 12–12, 2011.
- [8] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 185–198. USENIX Association, 2013.
- [9] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 285–300. USENIX Association, 2014.
- [10] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. Hug: Multi-resource fairness for correlated and elastic demands. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation*, pages 407–424, 2016.
- [11] Emilio Coppa and Irene Finocchi. On data skewness, stragglers, and mapreduce progress indicators. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 139–152. ACM, 2015.
- [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 ACM Symposium on Operating Systems Principles*, pages 153–167. ACM, 2017.
- [13] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware scheduling in Eagle: Divide and stick to your probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 497–509. ACM, 2016.
- [14] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In Shan Lu and Erik Riedel, editors, *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 499–510. USENIX Association, 2015.
- [15] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 48, pages 77–88. ACM, 2013.
- [16] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 42, pages 127–144. ACM, 2014.
- [17] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the 6th ACM Symposium on Cloud Computing*, pages 97–110. ACM, 2015.
- [18] Yihui Feng, Zhi Liu, Yunjian Zhao, Tatiana Jin, Yidi Wu, Yang Zhang, James Cheng, Chao Li, and Tao Guan. Scaling large production clusters with partitioned synchronization. In *Technical Report (http://www.cse.cuhk.edu.hk/~jcheng/papers/parsync_atc21.pdf)*. CUHK, 2021.
- [19] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, volume 11, pages 24–24, 2011.
- [20] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 99–115, 2016.
- [21] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the ACM SIGCOMM Computer Communication Review*, volume 44, pages 455–466. ACM, 2015.

- [22] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 65–80, 2016.
- [23] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 81–97, 2016.
- [24] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In David G. Andersen and Sylvia Ratnasamy, editors, *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. USENIX Association, 2011.
- [25] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, volume 11, pages 22–22, 2011.
- [26] Yuzhen Huang, Yingjie Shi, Zheng Zhong, Yihui Feng, James Cheng, Jiwei Li, Haochuan Fan, Chao Li, Tao Guan, and Jingren Zhou. Yugong: Geo-distributed data and job placement at scale. *Proc. VLDB Endow.*, 12(12):2155–2169, 2019.
- [27] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 261–276. ACM, 2009.
- [28] Tatiana Jin, Zhenkun Cai, Boyang Li, Chengguang Zheng, Guanxian Jiang, and James Cheng. Improving resource utilization by timely fine-grained scheduling. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 20:1–20:16. ACM, 2020.
- [29] Prajakta Kalmegh and Shivnath Babu. Mifo: A query-semantic aware resource allocation policy. In *Proceedings of the 2019 ACM International Conference on Management of Data*, pages 1678–1695. ACM, 2019.
- [30] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Annual Technical Conference*, pages 485–497, 2015.
- [31] James E Kelley Jr and Morgan R Walker. Critical-path planning and scheduling. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 160–173. ACM, 1959.
- [32] Kubernetes. Taking network bandwidth into account for scheduling. <https://github.com/kubernetes/kubernetes/issues/16837>, 2015.
- [33] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36. ACM, 2012.
- [34] Libin Liu and Hong Xu. Elasecutor: Elastic executor scheduling in data analytics systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 107–120. ACM, 2018.
- [35] Kay Ousterhout, Aurojit Panda, Josh Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The case for tiny tasks in compute clusters. In Petros Maniatis, editor, *14th Workshop on Hot Topics in Operating Systems, HotOS XIV, Santa Ana Pueblo, New Mexico, USA, May 13-15, 2013*. USENIX Association, 2013.
- [36] 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. ACM, 2013.
- [37] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient queue management for cluster scheduling. In Cristian Cadar, Peter R. Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues, editors, *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 36:1–36:15. ACM, 2016.
- [38] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmerek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: workload autoscaling at google. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo Seltzer,

- editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 16:1–16:16. ACM, 2020.
- [39] Malte Schwarzkopf, Andy Konwinski, Michael Abdel-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek, editors, *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 351–364. ACM, 2013.
- [40] Xiaoyang Sun, Chunming Hu, Renyu Yang, Peter Garraghan, Tianyu Wo, Jie Xu, Jianyong Zhu, and Chao Li. Rose: Cluster resource scheduling via speculative over-subscription. In *2018 IEEE 38th International Conference on Distributed Computing Systems*, pages 949–960. IEEE, 2018.
- [41] 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 Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 30:1–30:14. ACM, 2020.
- [42] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schüpbach, and Bernard Metzler. Albis: High-performance file format for big data systems. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 615–630. USENIX Association, 2018.
- [43] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Tetrished: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the 11th European Conference on Computer Systems*, page 35. ACM, 2016.
- [44] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Saha Bikas, 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*, page 5. ACM, 2013.
- [45] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In Laurent Réveillère, Tim Harris, and Maurice Herlihy, editors, *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, pages 18:1–18:17. ACM, 2015.
- [46] Yi Yao, Han Gao, Jiayin Wang, Ningfang Mi, and Bo Sheng. Opera: opportunistic and efficient resource allocation in Hadoop YARN by harnessing idle resources. In *Proceedings of the 25th International Conference on Computer Communication and Networks*, pages 1–9. IEEE, 2016.
- [47] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.

Fighting the Fog of War: Automated Incident Detection for Cloud Systems

Liqun Li[‡], Xu Zhang[‡], Xin Zhao^{§*}, Hongyu Zhang^{*}, Yu Kang[‡], Pu Zhao[‡], Bo Qiao[‡],
Shilin He[‡], Pochian Lee[◊], Jeffrey Sun[◊], Feng Gao[◊], Li Yang[◊], Qingwei Lin^{‡,†},
Saravanakumar Rajmohan[◊], Zhangwei Xu[◊], and Dongmei Zhang[‡]
[‡]Microsoft Research, [◊]Microsoft Azure, [◌]Microsoft 365,
^{*}The University of Newcastle, [§]University of Chinese Academy of Sciences

Abstract

Incidents and outages dramatically degrade the availability of large-scale cloud computing systems such as AWS, Azure, and GCP. In current incident response practice, each team has only a partial view of the entire system, which makes the detection of incidents like fighting in the “fog of war”. As a result, prolonged mitigation time and more financial loss are incurred. In this work, we propose an automatic incident detection system, namely Warden, as a part of the Incident Management (IcM) platform. Warden collects alerts from different services and detects the occurrence of incidents from a global perspective. For each detected potential incident, Warden notifies relevant on-call engineers so that they could properly prioritize their tasks and initiate cross-team collaboration. We implemented and deployed Warden in the IcM platform of Azure. Our evaluation results based on data collected in an 18-month period from 26 major services show that Warden is effective and outperforms the baseline methods. For the majority of successfully detected incidents (~68%), Warden is faster than human, and this is particularly the case for the incidents that take long time to detect manually.

1 Introduction

Reliability is a key quality attribute of large-scale cloud systems such as AWS, Azure, and Google Cloud Platform (GCP). Although tremendous effort has been devoted to improving reliability, cloud systems still suffer from incidents and outages [8, 10, 11]. Monitoring is widely used by cloud systems to check their runtime status. A monitoring system collects, processes, and aggregates quantitative data about a cloud system. When a system problem occurs, an alert is sent to an on-call engineer, who is expected to triage the problem and work toward its mitigation. A severe enough alert (or a group of aggregated alerts) is escalated as an *incident*.

*Work done during Xin Zhao’s internship at Microsoft.

†Qingwei Lin is the corresponding author of this work.

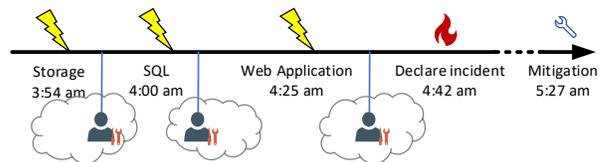


Figure 1: The timeline of handling a cross-service incident, where it took nearly 50 minutes to understand the situation and declare the incident.

Timely incident management is the key to reduce the system downtime. However, according to our experience, a reactive and ad-hoc incident detection is often employed in practice, which hinders effective incident management. We motivate our work using a real-world example. Fig. 1 shows the timeline of an incident caused by a flawed configuration change in the Storage service. The failed storage accounts affected several SQL databases, and the failure was further propagated to Web Application instances that were depending on the impaired databases. The failure triggered cascading alerts for Storage, SQL, and Web Application at 3:54 am, 4:00 am, and 4:25 am, respectively. After many rounds of discussions (which took nearly 50 minutes), the engineers finally realized that this was a cross-service issue, and an incident was declared. An experienced Incident Commander (IC) [4, 6] was then engaged to coordinate the mitigation process. Eventually, at 5:27 am, the incident was mitigated and all services were back to normal.

In this work, we focus on automatic incident detection for cloud systems, so that incidents could be declared and mitigated as early as possible. Incident declaration turns the mitigation process from chaotic to managed, especially for issues that require cross-team collaboration: the mitigation tasks are prioritized, teams are aligned, and customer impact is reduced. Our study is based on the Incident Management (IcM) platform of Microsoft Azure, one of the world-leading cloud computing platform. Though different companies [1–3]

implement slightly different incident response processes, we argue that they are essentially similar.

Incident detection is challenging. For a cloud system with extraordinary complex dependency among thousands of components, the on-call engineers, like in the fog of war, usually only have a partial view of the big picture. In the above-mentioned example, the engineers handling the storage alert could only see the affected storage accounts, but it was time-consuming to understand who was impacted. For engineers in the SQL and Web Application teams, it also took them a considerable amount of time to understand that they were affected by the underlying services. According to our interview with on-call engineers from the several service teams, such a diagnostic process could take as long as an hour. Also, there were a large number of concurrent alerts, making the situation even more challenging. Better communication processes/protocols can help, but will not solve the problem completely.

In this work, we propose Warden for effective and timely incident detection in IcM. Warden detects alerts that can potentially turn into incidents by employing a data-driven approach. A simple rule-based approach with human knowledge is insufficient in practice as it can be easily overwhelmed by a massive number of complex and ever-changing rules in a large-scale production cloud system. An example rule is “once a Storage alert was immediately followed by a SQL alert and they happened in the same datacenter, it is likely that they were related and constituted an incident”.

We develop a machine learning model to detect the incidents, and point out incident-indicating alerts for the on-call engineers. We have evaluated Warden using data collected in an 18-month period from 26 major services on Azure. According to our experimental results, Warden is effective and outperforms the baseline methods. Warden is also faster than human for the majority of successfully detected cases (~ 68%).

Our major contributions are summarized as follows:

- We study the problem of incident detection in the incident management (IcM) platform of Microsoft Azure. We have identified the obstacles that result in prolonged incident declaration time.
- We propose Warden, a framework to automatically detect incidents based on alerts in IcM. Warden quickly detects incidents and assists in task prioritization and cross-team collaboration.
- We evaluate Warden with real-world data collected from 26 major services on Azure. The evaluation results confirm the effectiveness of the proposed approach. We have successfully deployed Warden in IcM.

2 Background and Problem Formulation

In this section, we briefly describe the basic concepts about alerts and incidents and then formulate the incident detection problem. To help explain the concepts and the problem, we perform an empirical study of 5 services (named Big5) of Azure, namely Compute, Storage, Networking, SQL DB, and Web Application. These 5 services are common to almost all cloud computing platforms.

Alert ID: 200603407	Title: Ongoing VM critical failures	Severity: High
Service: Compute	Team: OS	Owner: Lily
Start Time: 2020-03-13 07:30:00	Mitigation Time: 2020-03-13 08:23:00	
Region: A	Data Center/Cluster (Optional): xxx/xxx	
Diagnosis logs	Monitor ID: DataCenterFailure	
Declare Time: 2020-03-13 07:52:00		Commander: Bob
Related Alerts		
Postmortem: summary, 5 why, timeline, root cause, repair items, etc.		

Figure 2: Main fields of an alert. Additional fields (grey colored) are appended when an alert is escalated into an incident. The escalated alert becomes the primary alert of the incident, while related alerts are manually linked.

2.1 Alerts and Incidents

Alerts represent system events that require attention, such as API timeouts, operation warnings, unexpected VM reboots, or network jitters. An alert consists mainly of fields shown in Fig. 2. Each alert has an impact starting time. Its mitigation time is filled when the problem is fixed. Alerts have different severity levels - low, medium, and high. Alerts are reported by monitors, which are continuously running programs that keep tracking the health status of a certain aspect of a service component. Each alert carries its monitor ID. Most alerts has its region information and some with more fine-grained location information such as datacenter or cluster.

Figure 3 shows the number of alerts per day for Big5 services in a one-year time frame. There are tens of thousands of concurrent alerts produced by a large number of components in the cloud system. Even for high severity alerts, the number could be hundreds or even several thousands. Figure 4 shows the number of active monitors per month. The number is gradually increasing as the platform scaling up and being improved.

In general, incidents are declared under severe situations. Often, problems taking a long time to solve or requiring cross-team collaboration are also declared as incidents. One incident usually triggers correlated alerts [21, 53]. An incident is thus

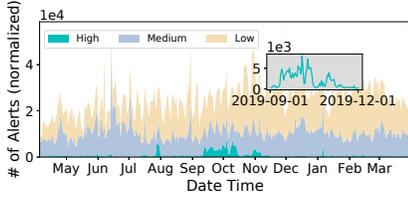


Figure 3: Number of alerts of different severity levels for the Big5 services.

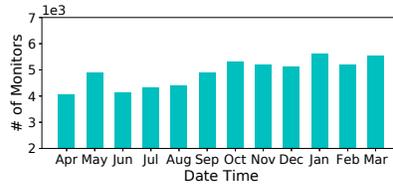


Figure 4: Number of monitors of the Big5 services actively reporting alerts per month.

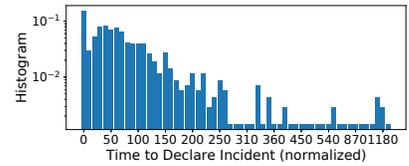


Figure 5: The histogram of time to declare incidents for the Big5 services.

escalated from one alert or a group of correlated alerts. Additional fields are added when alerts are escalated into incidents as shown in Fig. 2. The *primary* alert of an incident is usually the one closest to the root cause or with the earliest alerting time. The incident commander and the on-call engineers manually link related alerts with the primary alert based on expert knowledge. In the one-year-length period, there were several hundreds of incidents caused by the Big5 services, among which $\sim 67\%$ are incidents impacting more than one services. These incidents are typically the hard ones for detection and mitigation due to the partial view issue.

Fast incident detection is critical for ensuring timely incident management. The time to declare incidents takes around one-third of the whole mitigation time in our study. Figure 5 shows the histogram of time to declare incidents for the Big5 services. We normalized the time to declare incidents in Fig. 5 to protect company sensitive data. The reader could use the public incident reports [8, 10, 11] as a reference for the incident mitigation time. About one-third of the incidents are declared within minutes. Most of these incidents are single service issues that could be straightforwardly detected based on rules. Nearly half (47.6%) of the incidents take more than 30 NTUs (normalized time units) to declare, and a long tail of incidents require even hours. We manually analyzed these cases and found that most of these cases fall into the category where a small issue slowly turns into a big one. Our proposed approach is thus helpful by detecting incidents from a global perspective.

2.2 Problem Formulation

We now formally define the problem of incident detection. We denote the current wall clock as t . The input is the set of alerts coming from different services within the time window \mathcal{W}_t^l whose duration is $(t - w, t]$, where w is the length of the window. One monitor can be triggered many times over its lifetime. The series of alerts reported by any monitor m is called an *alert signal*, denoted as s^m .

From the input time window, we want to estimate the probability of $P_t = P(I_t | \mathcal{A}_t)$, where $I_t \in \{0, 1\}$ is the indicator of incident. \mathcal{A}_t is the observed set of alert signals in the time window that ends at t . We then infer that an incident has occurred

by examining whether P_t surpasses a threshold.

In addition, we need to extract a subset of alert signals which are indicating the incident from the time window, denoted as $\hat{\mathcal{A}}_t \subset \mathcal{A}_t$. System failures often lead to correlated alert signals. The physical meaning is that each alert signal $s^m \in \hat{\mathcal{A}}_t$ therein is one symptom caused by the underlying incident. We name $\hat{\mathcal{A}}_t$ as *incident-indicating* alert signals. According to our definition, we shall have $P_t = P(I_t | \mathcal{A}_t) = P(I_t | \hat{\mathcal{A}}_t)$.

In summary, we divide the incident detection problem into two steps. The first step is incident detection based on the observed alert signals from different services in a recent time window. The second step is to understand which alert signals are likely caused by the detected incident, i.e., identify the incident-indicating alert signals. When we detect the incident, we only notify the relevant on-call engineers. We describe the details of our solution in the next section.

3 The Proposed Approach

The workflow of Warden is shown in Fig. 6, which consists of the following major steps:

Alert signal selection Monitoring a very large-scale system is challenging due to the sheer number of components being watched. The raw alert data is not only large in volume but also contains noise. Therefore, we select only a subset of monitors which exhibit relatively strong association with incidents. Details about alert signal selection could be found in Sec. 3.1.

Incident Detection We adopt a data-driven paradigm for incident detection. We carefully extract a set of features from alerts in a recent time window based on domain knowledge in Sec. 3.2.1. Then, the feature vector is fed to a Balanced Random Forest (BRF) [37] model constructed offline. With this model, we could detect the occurrence of incidents. Section 3.2.2 explains how we train and apply the model.

Incident-indicating alerts identification We developed a novel approach to identify the incident-indicating alerts. We start by dividing the alert signals into groups. Then, we assign

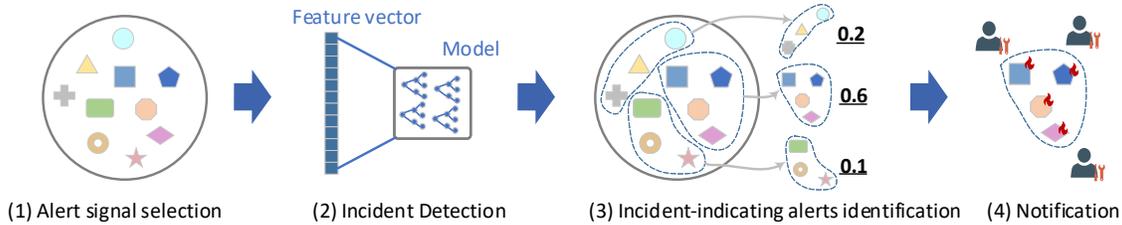


Figure 6: (1) Recent alert signals in a time window are selected and cached. (2) A feature vector is extracted and fed to a model for incident detection. (3) We extract a group of alert signals which most significantly indicate the detection result. (4) Relevant on-call engineers are notified for further investigation.

a score to each group based on an algorithm of our design called Group Shapely Value (GSV). The group of alert signals with the highest importance score is identified as the incident-indicating alerts. We explain the grouping and group-based model interpretation in Sec. 3.3. This approach is inspired by model interpretation [43], which is about understanding the relationship between inputs and outputs of a machine learning model. However, our goal is to extract a group of alerts indicating the detection result as shown in Fig.6 (3). The difference between incident-indicating alert identification and model interpretation is discussed in Sec. 5.

Notification We integrated our system into the IcM of Azure. A notification is pushed to all involved alerts when a potential incident is detected. When the corresponding on-call engineers are aware of the detected emerging issue, they could better understand the big picture and form a collaboration group to declare and resolve the incident. We introduce the practical usage scenario in Sec. 3.4.

3.1 Alert Signal Selection

Feeding alerts from all monitors could overwhelm the detection model. Figure 3 and Figure 4 show the number of active monitors and produced alerts for the Big5 services, respectively. There are even more monitors and alters considering other services. In contrast, the number of incidents is relatively small.

The purpose of alert signal selection is to identify a subset of monitors which emit alerts exhibiting a relatively high correlation with incidents. For this purpose, we calculate a score for each monitor. The higher the score, the more likely the alert signal from this monitor will indicate that the cloud is experiencing an incident. If we observe that the alerts from a certain monitor co-occur with incidents frequently, we should assign a high score to that monitor. With this intuition, we adopt the Weighted Mutual Information (WMI) [26]. WMI is a measure of the mutual dependence between two variables, i.e., quantifying the “amount of information” obtained about one random variable through observing the other random

variable [50].

Formally, denote I as the set of all incident. For each monitor m , we could calculate the WMI between the alert signal s^m and all incidents, i.e., $I(s^m; I)$. One monitor is usually only effective in detecting a specific problem. However, there are many different types of incidents. Therefore, calculating the WMI of one monitor against all incidents is not appropriate in characterizing its effectiveness, especially given one incident type may have far more cases than another type. Yet, a minority incident type may be equally, if not more, important than a majority type.

We divide all incidents into different subtypes based on their owning team that handled this incident. We notice that, in a production organization, each team usually focuses on specific functionalities [17, 18, 30]. Therefore, we assume that all incidents handled by the same team are similar, and therefore, constitute a subtype of incidents. We use $i \subset I$ to denote the a subtype of incidents. Then, the WMI between s^m and i could be calculated in Eq. (1).

$$I(s^m; i) = \sum_{s^m} \sum_i w(s^m, i) \cdot P(s^m, i) \cdot \log \left(\frac{P(s^m, i)}{P(s^m)P(i)} \right) \quad (1)$$

Both s^m and i are series of events with $s^m, i \in \{0, 1\}$. Eq. (1) essentially elaborates all value combinations of the two variables and examines their inherent dependence. We divide historic data into windows in order to calculate Eq. (1). For instance, to derive the joint probability $P(s^m = 1, i = 1)$, we simply count the number of windows containing both events. This value is then divided by the total number of windows. Other joint and marginal probabilities could be derived in a similarly way. The coefficients $w(s^m, i)$ in Eq. (1) are used to assign different weights to different value combinations.

Once we have the WMI for every pair of monitor and incident subtype, we can calculate the sum of score for m by going through all incident subtypes. We use this score to rank the monitors and pick the top 150, which are most effective for incident detection. We will evaluate the number of selected monitors in Sec. 4.6.2. The alerts not reported by the selected monitors are ignored by our system.

3.2 Incident Detection

We formulate the incident detection problem as a binary classification problem. The input consists of alert signals in a time window. A classifier is trained offline based on the historic labelled samples. Then, we apply the trained model for online incident detection.

3.2.1 Feature Extraction

We identify the following features from alerts generated by selected monitors in a time window:

Alert signals: The first set of features characterize the alert signals. Each alert signal is a series of alerts generated by a certain monitor within the time window.

- Alert count. In the event of a malfunction, the alert signal is typically stronger than usual. As a result, we count the total number of alerts and the high-severe alerts for each monitor, respectively. Multiple services may start to report alerts, and we, therefore, count the number of alerts from each service. Besides, we count the total number of alerts, the total number of monitors reporting high-severe alerts, and the total number of services, respectively.
- Alert burst. We adopt a time window length of 3 hours¹. Our window size is large enough to accommodate related alerts. However, under certain conditions, alerts erupt only in a small time and space. For instance, a partial power failure may hit only a few racks in a datacenter. To capture this feature, we calculate the max number of monitors reporting alerts and the max number of high-severity alerts in any datacenter, respectively. Likewise, in the temporal dimension, we calculate the two features for all child sliding windows of 10/30/60 minutes in length, respectively.

Engineer activities: Engineers leave their footprints when they are working on alerts in IcM. The second set of features characterize the intensiveness of engineer activities. Engineers could post their diagnosis logs on the alert page. They could also launch a bridge meeting for online discussion. The bridge meetings are hosted by IcM and attached to the alerts. If people in one service want to reach out to another service, they could use the notification tool in IcM which automatically routes them to the on-call team in the other service. Though private communications (via personal phone calls or IM apps) still exist, most traffic today is through IcM.

- Diagnosis log count. The engineers tend to leave diagnosis logs on the alert web pages as well as in the bridge meetings. We count the number of discussion posts for

¹According to our empirical study, over 80% alerts are triggered within 3 hours after the impact start time.

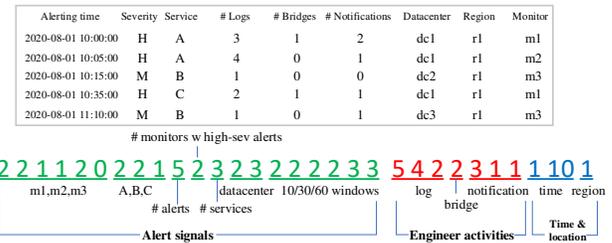


Figure 7: An example of feature extraction with 5 alerts in the time window generated by 3 monitors from 3 services. One feature vector is constructed for each time window.

each alert and the number of concurrent bridge meetings in IcM, respectively.

- Notification count. Notifications are sent to initiate cross-team collaborations. We thus count the number of notifications within the time window.

In addition to the aforementioned features, we also include features about time (day and hour) and location (Region ID). We illustrate the process of feature extraction using a dummy example of 5 alerts in Fig. 7. We show a few example columns of the derived feature vector.

3.2.2 Model Construction

We construct data samples using alert data with a sliding window to training an incident detection model. The window size is set to 3 hours as discussed in the previous section. The window is sliding with a stride length of 5 minutes, which is small enough in order not to miss any significant changes. The label of the sliding windows overlapping with incidents is positive; otherwise, negative. The labeling methodology is illustrated in Fig. 8. The two vertical bars represent the impact starting time and the mitigation time of the incident, respectively. The example sliding window (i.e., the shadowed rectangle) in Fig. 8 is thus labeled as positive. The 5-minute stride length is only used for offline model construction. For online usage, Warden runs once every minute. More details could be found in Sec. 3.4.

Not all alert signals in the window are part of the incident. In Fig. 8, only the alerts marked by the dashed curve are symptoms of the incident. This group of alert signals is $\hat{\mathcal{A}}_t$ according to our definition in Sec. 2.2. In our evaluation, we use the links between alerts to obtain the relationship between an incident and its related alerts. The links were manually labelled by on-call engineers or the incident commander during the incident mitigation process. We note that the related alerts are only available for historic data. For online detection, this information is unknown (or partially unknown).

Once we have the extracted features and the labels, we train a model and apply it for online incident detection. In our

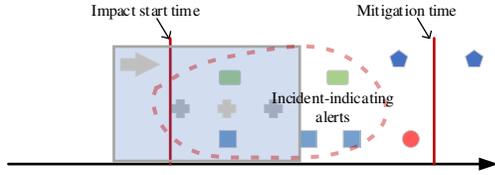


Figure 8: The sliding window is labelled as positive if any incident impact start time falls into it. Those time windows with no overlap with incident impact time ranges (from impact start to mitigation) are labelled as negative.

context, the number of incidents is much smaller than that of alerts. We therefore choose the BRF [37] classifier because of its robustness to noise and ability to handle imbalanced data. Warden is a generic framework, and any model could be plugged in for the classification task. More details about the model selection are presented in Sec. 4.6.1.

The output of the model is the probability $P_t \in [0.0, 1.0]$. We use a confidence threshold as discussed in our problem formulation in Sec. 2.2. If the confidence is higher than the threshold, we believe a potential incident is happening.

3.3 Identifying the Incident-indicating Alerts

We need to inform the right people when a potential incident is detected, and otherwise, it is non-actionable. In other words, we need to find out which alerts in the window are related to the detected incident, as illustrated in Fig. 8. After we identify these incident-indicating alerts, we notify their responsible on-call engineers.

It is a non-trivial task given many concurrent alert signals in the window. We propose a novel approach inspired by two intuitions: first, the target alert signals must be a group of correlated alert signals due to the same reason; second, the target alert signals should contribute significantly to the detection result. Such a group of correlated alerts is indicative to the detected incident.

Our approach has two steps. We first group alert signals that are likely related to each other into signal groups. Then, we interpret the incident detection model on a group basis.

3.3.1 Alert signal grouping

Correlated alert signals could be identified based on statistics on their co-occurrence history. We put two alert signals into the same group under one of the two conditions: (1) they co-occur frequently in history; (2) they fire from the same cluster.

An alert signal is a series of events. We thus first characterize the correlation between a pair of alert signals as we did for monitor selection in Sec. 3.1. We use a threshold to tell if one is strongly correlated with another. In addition, we

conduct statistics on related alerts for historic incidents. For each incident, people manually link related alerts. If two alert signals are frequently linked in history (more than 3 times), we believe that they are correlated if they present in the same time window.

Then, if two alert signals fire in the same cluster, we believe that they are correlated. In a large-scale cloud computing platform like Azure, a cluster is usually dedicated for a certain functionality (e.g., for storage, compute, or database). If two alert signals emerge from the same cluster in a short time frame, they are inherently related in a high probability.

We apply the aforementioned rules to group alert signals in the current time window. After grouping, the alert signals are divided into corresponding groups, as illustrated in Fig. 6.

3.3.2 Group-based Model Interpretation

We now discuss how to obtain the incident-indicating alerts. The basic idea is to rank the signal groups according to their contribution to the prediction result of the detection model. Then, we pick the top group of alert signals.

For this purpose, we develop a novel algorithm called Group Shapely Value (GSV). GSV is heavily inspired by the Shapely Value method [47] which is a model-agnostic interpreter.

Algorithm 1 Group Shapely Value (GSV)

Require:

- The classification model, M ;
- A testing window, \mathcal{W}_t ;
- The alert signal groups set associated with \mathcal{W}_t , \mathcal{G}_t ;
- The target group, $g \in \mathcal{G}_t$;
- Background training data, B
- Sampling rounds, n

Ensure:

- The Shapely Value c for target group g ;
 - 1: $\phi = 0$
 - 2: **for** 1 to n **do**
 - 3: Randomly select a subset \mathcal{G}' , $\mathcal{G}' \subseteq \mathcal{G}_t$
 - 4: $\mathcal{G}_{other} = \mathcal{G}_t \setminus (g \cup \mathcal{G}')$
 - 5: Randomly select a sample $x_b \in B$
 - 6: $b_1 = x_t(\mathcal{G}_{other}) \vee x_t(g) \vee x_b(\mathcal{G}')$
 - 7: $b_2 = x_t(\mathcal{G}_{other}) \vee x_b(g) \vee x_b(\mathcal{G}')$
 - 8: $\phi = \phi + M(b_1) - M(b_2)$
 - 9: **end for**
 - 10: $c = \frac{\phi}{n}$
-

The original Shapely Value method only estimates the importance for each feature value. In contrast, GSV is able to interpret the model for each alert signal group.

The intuition behind GSV is that we assemble combinations of alert signal groups to evaluate the contribution from each individual group. We leverage the Monte Carlo approximation method [47] to overcome the combination explosion

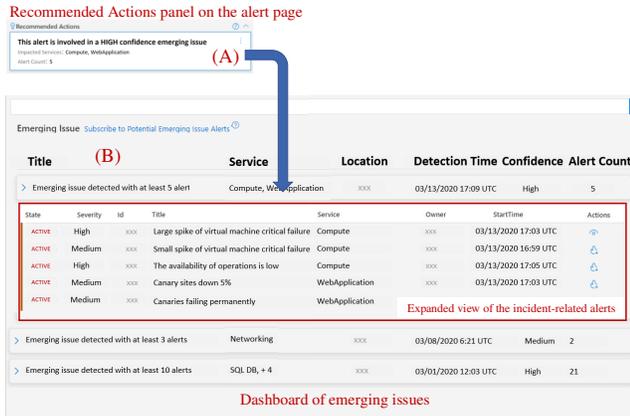


Figure 9: Web-based UI of Warden in IcM. (A) shows the “Recommended Actions” panel and (B) shows the “Emerging Issues” dashboard.

problem. The details of GSV can be referred in Algorithm 1. The input of the algorithm are the model M and the set of alert signal groups in the time window denoted as G_t obtained in the previous grouping step. GSV requires a set of background data B , which is randomly sampled from the training data. n is a hyper-parameter which is typically set to several hundred.

In Algorithm 1, we construct two synthetic samples b_1 and b_2 in each iteration, as shown in Line 6 and Line 7, respectively. Then, we calculate their difference in prediction probabilities by the model M . The operator $x(\cdot)$ retains only the feature values from a subset of alert signal groups of the testing sample x . Intuitively, the difference between b_1 and b_2 indicates the significance of the current value of g (i.e., $x_t(g)$) given a background sample $x_b \in B$. The iteration continues for n rounds and the accumulated ϕ is normalized to c . We use c as the score for the group g . We pick the group with the highest score as the incident-indicating alert signals.

3.4 Using Warden in Practice

To facilitate using Warden in practice, we develop a Web-based UI as shown in Fig. 9. The UI consists of two parts: (A) a “Recommended Action” panel located on each alert page, and (B) a dashboard of “Emerging Issues” for all detected incidents. Each detected incident is with a list of incident-indicating alerts.

Warden runs periodically at a fixed interval (in our practice, Warden runs once every minute, and the main task takes around 20 seconds). When Warden detects a potential incident, a notification is pushed to the “Recommended Action” panels of all identified incident-indicating alerts. Our notification is basically saying: *this alert is of high priority and it may be part of an emerging incident. Please check other alerts are likely triggered by the same issue as well.* When the engineer sees the notification pops up in the “Recommended Action” panel, she or he could click to jump to the dashboard for

more details. The list of related alerts on the dashboard helps the engineers to understand the big picture, prioritize their work, and initiate cross-team collaboration. We will describe real-world cases in Sec. 4.7.

4 Experimentation

In our evaluation, we aim to answer the following research questions:

- RQ1: How effective is Warden in detecting incidents?
- RQ2: How fast can Warden detect incidents compared with human?
- RQ3: How accurate can Warden extract incident-indicating alert signals?
- RQ4: What is the impact of key system settings on incident detection performance?

4.1 Dataset

To evaluate Warden, we collect alert data ($\sim 240G$) from the IcM of Azure within a 18-month-length period, starting from Oct.2018. We carefully selected 26 major services of Azure, including the Big5. Hundreds of people in dozens of teams are behind each service. The total number of incidents reported by the 26 services accounts for $\sim 72\%$ of all incidents in Azure. The dataset includes over 10 million alerts. We use the data in the last two month for testing and the previous months for training. Our training and testing data contain around 82% and 18% of all incidents, respectively. We organize our dataset into sliding windows and conduct data labeling as discussed in Sec. 3.2.2. The positive-to-negative ratio is around 1:25. More details about our dataset could be found in Table 1.

	Training	Testing
Low-severity alert	31.4%	7.8%
Med-severity alert	51.6%	6.3%
High-severity alert	2.6%	0.3%
Incident	82.2%	17.8%

Table 1: Details about the dataset for experimentation.

4.2 Evaluation Metrics

Warden runs on a fixed interval. Once a potential incident is inferred, Warden notifies related on-call engineers. The metric we care most about is the accuracy of incident detection and incident-indicating alert identification. For incident detection, we calculate the precision, recall, and F1-score. Precision measures the ratio of the sliding windows identified by Warden are truly containing incidents. Recall measures

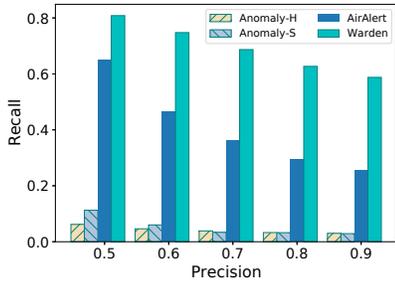


Figure 10: The recall of Warden and three baselines recorded with precision from 0.5 to 0.9.

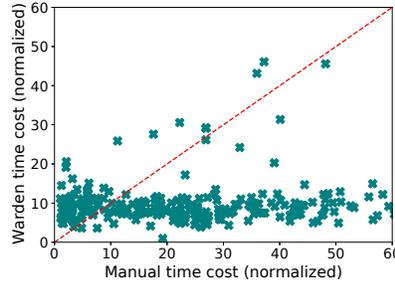


Figure 11: The time taken by manual incident declaration versus that by Warden.

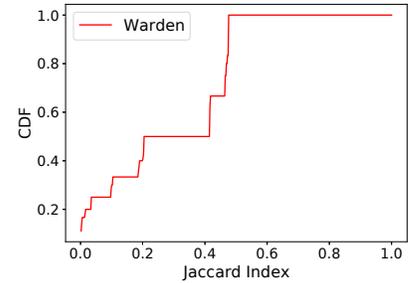


Figure 12: The CDF of Jaccard index between the owning services of the identified alerts and the actual impacted services.

the ratio of positively labeled windows recognized by Warden. F1-score is the harmonic mean of precision and recall. We also calculated AUC-PR in model selection, which is the area under the precision-recall curve. For incident-indicating alerts identification, we use Jaccard Index [33] to measure the difference between the identified group of alerts versus the manually linked alerts.

4.3 RQ1: Incident Detection Effectiveness

To our best knowledge, Warden is the only deployed system for incident detection based on monitor reported alerts in large-scale cloud systems. To better evaluate the effectiveness of incident detection, we compare Warden with the following baselines:

Anomaly detection Anomaly detection is a widely used approach to discover system faults. An incident is detected if the actual number of alerts is significantly higher than the prediction. We implement our baselines based on Prophet [49] from Facebook, which is widely adopted for time-series forecasting. The first baseline, namely **anomaly-H**, is evaluated on high-severity incidents; the second, namely **anomaly-S**, is evaluated on incidents reported by our selected monitors.

AirAlert AirAlert [23] uses a XGBoost [20] model to predict outages with only the alert count features from each monitor. We implemented AirAlert with input from our selected monitors.

The comparison results of Warden with the three baselines are presented in Fig. 10. There is a trade-off between precision and recall for a certain model. As discussed in Sec. 3.2.2, we use a confidence threshold to tune our model. In Fig. 10, we tune the models to change their precision to different values (0.5 ~ 0.9) and record the corresponding recall values. The x-axis of Fig. 10 is the precision and the bar charts show the corresponding recall values. The results show that Warden is effective and outperforms the baseline methods.

The anomaly detection based approaches achieve very low recall. The key reason is that an incident does not necessarily cause a spike, especially given the large volume of alerts. Only those incidents with extremely large impact (usually referred to as outages) could lead to high peaks. The anomaly-S achieves slightly higher recall than anomaly-H, which demonstrates the effectiveness of our monitor selection. AirAlert achieves better performance than the anomaly detection based approaches. As discussed in Sec. 3.2.1, we identified more features in addition to alert count. With the full set of features, Warden significantly outperforms AirAlert. This evaluation result justifies the effectiveness of feature engineering.

False alarms could be annoying to deal with in real-world scenario, as they can cause incorrect task prioritization or unnecessary communication. To avoid spamming the engineering team with false alarms, we ensure a precision of above 90%, which corresponds to a recall of ~ 58%. The achieved F1-score is 0.71.

4.4 RQ2: Fast Incident Detection

We compare the detection time using Warden and the manual incident declaration time for incidents in our test dataset. For all successfully detected incidents, Warden is able to perform detection faster than human in ~ 68% of cases. The time saving for an incident is measured between the notification time reported by Warden and the incident declaration time reported by the on-call engineers. The median time saving for these cases is 21.8 NTUs, which accounts for ~ 15% of the whole time-to-mitigate for these incidents. The readers could use the public incident reports [8, 10, 11] as a reference for incident mitigation time.

Figure 11 compares the time taken by manual incident declaration and that by Warden. Each cross in the figure represents one case in our testing dataset. It is clear Warden beats human for majority of cases. Moreover, we can see that Warden can particularly help reduce the incident detection time for those cases performed poorly by human. In practice, for those cases when Warden falls behind human, we simply

ignore them without sending notifications. To make the figure more compact, we only show data points within 60 NTUs.

4.5 RQ3: Accuracy of Extracting Incident-indicating Alert Signals

It is tricky to evaluate how accurately can Warden extract the incident-indicating alert signals from the current time window. Our ground truth is based on manually linked alerts in incidents. As discussed in Sec. 2.1, the incident commander and on-call engineers manually link related alerts to the primary alert of an incident. Since this is done manually, it is easy to miss some related alerts. According to our empirical study, usually, only partial alerts are linked during a failure event. Therefore, it is infeasible to rely on the manual links for evaluation. However, we discovered that the impacted services are more reliable. When an incident occurs, more than one alert usually gets triggered for each impacted service. People usually selectively pick one or a few alerts and add to the primary alert, which are used to track the health status of that service. As a result, we use only the group of impacted services, instead of the linked alert signals, for evaluation.

We extract the incident-indicating alerts from the current time window as described in Sec. 3.3. For evaluation, we compare the owning services of the identified alert signals with the actual impacted services. Figure 12 shows the CDF of the Jaccard indices for the cases recalled by Warden in our testing dataset. First of all, we can see that Warden achieves a perfect alignment with the ground truth for 53% cases. For 78% cases, the Jaccard index is above or equal to 50%. Moreover, the curve in Fig. 12 is constantly above 0, which means that Warden can find at least one impacted service when incidents have been detected. In general, Warden can accurately find the impacted services inside the current time window.

4.6 RQ4: The Impact of Key System Settings

So far we have evaluated our system with a fixed set of settings. We now address the question of how different system settings affect the performance. Specifically, we examine three sub-questions: (1) What is the impact of different classification models on incident detection? (2) how many monitors should we select? and (3) how much data and how often is required to train our model?

4.6.1 Classification Model Selection

Warden uses a classification model for incident detection. We evaluated a few popular candidates, namely Penalized Linear Regression (PLR) [23], SVM [24], Balanced Random Forest (BRF) [37], and LightGBM [36]. To account for data imbalance, we assign different class weights to positive/negative samples for PLR, SVM, and LightGBM; for BRF, we adjust

the sampling rate. These settings are all tuned with grid search using validation data. The results are shown in Table 2.

Model	F1	Recall	Precision	AUC-PR
PLR	0.505	0.361	0.840	0.64
SVM	0.604	0.655	0.561	-
LightGBM	0.680	0.658	0.704	0.71
BRF	0.704	0.658	0.757	0.73

Table 2: The comparison results of different models.

We mainly use the AUC-PR for comparison. Since AUC-PR does not apply for SVM, we also record the F1-score and the corresponding precision and recall. BRF achieves the highest AUC-PR of 0.73 compared with PLR and LightGBM. For BRF and SVM, we could see BRF achieves higher precision and recall. In conclusion, BRF outperforms the other candidates, and therefore we adopt BRF in our system.

4.6.2 Alert signal selection

In Sec. 3.1, we have derived a score for each monitor and ranked all monitors accordingly. One remaining question is how many monitors should be selected. Intuitively, selecting more monitors will improve the coverage by detecting more incidents. However, more monitors also increase the complexity of the model as well as the data volume. Therefore, we try to find the minimum set of monitors while achieving satisfactory precision and recall.

We measure the recall at different precision values (0.6, 0.7, and 0.8) varying the number of monitors. The results are presented in Fig. 13, where the x-axis denotes the number of monitors and the y-axis shows the recall. The three curves are the recall values with different precision values, respectively. The number of monitors is ranging from 25 to 300.

We can see that the recall rises significantly for all three curves, when the number of monitors increases from 25 to 150, which is in line with our expectation. This observation indicates that a moderate fraction of incidents could be detected with a relatively small number of monitors. Then, increasing the number (when exceeds 150) would not increase the coverage as the newly-added monitors become less indicative of incidents. On the other hand, adding more monitors will increase the number of alerts to process. As a result, we set the number of selected monitors to 150 in our system.

4.6.3 Training data and frequency

We also evaluate our model with different training data lengths. Specifically, we keep the later two months in our dataset for testing and vary the length of training data from 1 to 16 months. For each round, we record the F1 score and the result is shown in Fig. 14.

The x-axis shows the length of training data from 1 to 16 months, and the y-axis is the F1 score. We can see that F1 rises

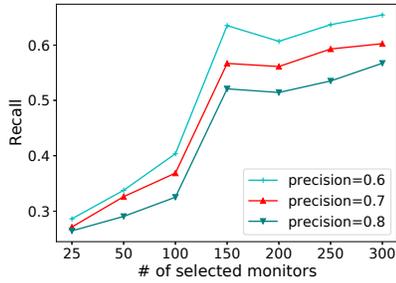


Figure 13: The recall at different precision values with different number of selected monitors.

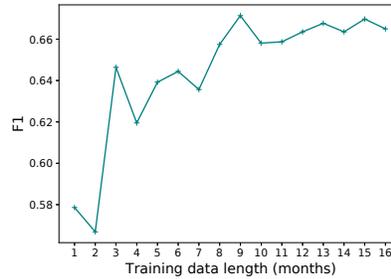


Figure 14: The F1 value with training data length from 1 to 16 months.

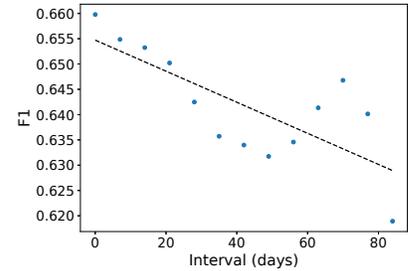


Figure 15: The F1-score with different training and testing data interleaving intervals.

with more training data. However, the gain becomes minor with the training data length exceeding 9 months. Therefore, for production usage, we collect data from the last 12 months for training, which is sufficient to reach optimal performance.

Another question is how often we need to retrain the model. We use 9 months of data for training and 2 months for testing. We interleave the training and testing data by m days, where m ranges from 0 to 80. We record the F1 scores with different interleaving intervals. Fig. 15 shows the result where the x-axis is the interleaving days and the y-axis is the mean F1 score. The dashed line is the regression fit which clearly shows a decreasing trend.

We can see a decline in performance as the data evolves (e.g., newly added monitors as shown in Fig. 4). In our system, we retrain our model on a weekly basis to capture incremental data changes. Our training is conducted on a node with 48 cores and 512 G memory. The training process with a 12-month dataset takes 16 cores and ~ 8 G memory, and costs ~ 1.2 hours.

4.7 Case Studies

Warden has been deployed online in IcM of Azure for around 3 months till May 2020. In this section, we describe two real-world cases.

Case 1: power failure incident The first case is an incident caused by a datacenter power failure. Counterintuitively, not all incidents caused by power issues are easy to detect. Few power issues lead to real incidents due to resource redundancy. In this case, the region of the datacenter was hit by an ice storm, resulting in a large-scale power failure. The affected racks lost power when their supporting UPS units drained. It took tens of minutes before running out of power supply.

Multiple services got affected gradually during this incident. Warden sent out an initial notification at receiving alerts from the Compute service. The alerts were about “Compute Manager QoS degradation”. Our notification immediately attracted the attention of the on-call engineer. When the engineer was investigating the problem, Warden gradually identi-

fied more than 10 alert signals (including “Virtual machines unexpected reboot”, “Web Application probe alert”, “TOR dead event”, “Too many partition load failures”, “API Unexpected Failures”, etc.) in following time intervals. Not only the inference confidence of the model was raised substantially, but also more on-call engineers from multiple teams were engaged. They were on the same page, and a bridge meeting was set up for collaborative problem-solving. Without Warden, the engineers in different service teams would have to run individual diagnoses and discuss back-and-forth for a prolonged time until they realized that they were impacted by a power failure.

Case 2: networking incident The second case is an incident caused by TOR (top-of-rack router) down. When the TOR device is down, a VM will lose the connection to its storage, resulting in an unexpected reboot. Meanwhile, the issue of VM unexpected reboot could also be caused by storage problems such as disk failures. The current practice is that on-call engineers run diagnostic tools to find out the root cause [51]. This diagnosis is time-consuming and requires cross-team collaboration.

In this case, a VIP customer’s service was impacted and timely root cause analysis (RCA) was required. When the TOR down alert was received, Warden notified the Networking on-call engineer immediately. Several minutes later, the alert from the Compute service arrived, and Warden correlated the two alerts together. When the RCA request came to the Compute team, it was immediately concluded the incident was caused by the networking issue. Timely RCA helped increase our customer’s confidence.

5 Discussions

Why can incidents be detected? Incidents can be caused by one-off issues like code bugs or flawed configurations. Once these issues get fixed, they might never happen again. However, the detection is not based on the root causes, but on symptoms (the triggered alerts) which indeed exhibits

certain repetitive patterns. In addition, some root cause issues indeed happen again and again, like hardware failures. For example, we examined the incidents caused by cluster-level partial power failures in history. The consequential impact vary slightly from one case to another. Therefore, it is also non-trivial to detect based on manually crafted rules. In conclusion, a data-driven approach is appropriate for incident detection.

Generalizability of the proposed approach Incident detection is a common problem for all cloud platforms. Different companies implement different incident response processes. For example, Azure has a 4-step incident management process as discussed in [17]. The steps are detection, triage, mitigation, and resolution. GCP employs a very similar 4-step process as described in [1], with more detailed child processes in each step. NIST proposed a high level of abstraction for incident response with similar steps in [5]. Warden relies on the monitoring system which is a key building block to ensure service reliability. Therefore, our proposed approach could be extended to all these incident response platforms.

Trade-off in detection accuracy and delay Warden aims to detect the incidents as early as possible. However, there is a trade-off between the accuracy and the delay. We can wait for long enough until the alerts accumulate so that we can accurately detect incidents. On the other hand, we can make detection with only early weak signals, which will save time but also lower our confidence. So far, the primary audience of our system are the on-call engineers. Therefore, we conducted user studies to understand their practical concern. We conclude a precision above 90% is satisfactory for real usage, and we then tune our system accordingly.

Difference between incident-indicating alert identification and model interpretation Identifying the incident-indicating alerts is not solely a model interpretation problem [43]. Algorithms, such as SHAP or LIME [40, 41, 45], rank the input features based on their contribution to the prediction result, without considering the inherent relationship between the features. In our system, the features are constructed from the alert signals and we want to understand the contribution of each alert signal, instead of individual features.

5.1 Threats to Validity

Subject system: We have limited our evaluations to 26 major services on Azure, which produce the majority of incidents. The services we pick are representative of large-scale cloud computing platforms. For example, the Compute, Networking, and Storage are the three most fundamental services. We also include SQL DB, Web Application, Backup, Data Pipeline, etc., which are common services provided by all platforms such as AWS, Azure, and GCP. The incidents in our dataset

are caused by a variety of common reasons. The top 5 deterministic reasons are code bug, network hardware issue, configuration, code defect, and design flaw. In our future plan, we will extend our evaluations with more services.

Label quality: We rely on manually linked alerts in our evaluation in Sec. 4.5. In practice, usually, only part of related alerts are linked correctly. The major problem is the missing links. Instead of manually linking all related alerts, the on-call engineers often only selectively link one alert for one service to track the cause-effect relationship. As a result, we have to evaluate the correctness of our extracted alerts on a service basis. Sometimes, people may even fail to involve a truly impacted service or add wrong links between irrelevant alerts. We incorporated experts from IcM to verify our label data and made a number of corrections. The label quality problem is thus greatly mitigated.

6 Related Work

Fault Detection and Localization: A significant amount of work [14, 28, 29, 31, 32, 34, 42, 44, 48, 51] exists on failure detection in cloud environments. Researchers [28, 31, 44] have advocated the scenario where minor failures from low-level services may cause large-scale impact to dependent services, which actually motivates our work of using alerts from multiple services to detect incidents. Some work [13, 25, 35, 42, 51] leverages the service dependency graph or API invocation chain to localize the root cause. Other work [14, 29, 32] proposes to use agents in distributed systems for failure detection and localization. In this work, we lay our foundation on top of the existing failure monitoring infrastructure built by individual services. We do not assume that we know the hierarchical dependency among components, which we believe is challenging for a large-scale dynamically changing cloud computing platform.

Time-Series Anomaly Detection: We notice that there is a body of work on anomaly detection and root cause localization for multi-dimensional time-series data [16, 38, 52]. There are plenty of commercial anomaly detection frameworks [7, 9, 12] offered by companies. In contrast, our work is based on alert data produced by cloud monitors. The alert data (described in Sec. 2.1) is very different from time-series metric data or console logs, and is commonly seen in cloud systems. Besides, our approach can not only detect the occurrence of cross-service incidents but also identify the incident-indicating alerts. We propose a novel algorithm called GSV for extracting the incident-indicating alerts. Finally, we target incidents that require cross-team collaboration (i.e., cross-service incidents). Therefore, we believe our work is novel and significantly different from the related work on time-series anomaly detection.

Incident Management: Haryadi et al. [27] analyze hundreds of officially published outages. Recently, there are also increasing interests in incident triage [17, 18, 39, 46, 56]. Jun-

jie et al. [19] use deep learning to find low severity incidents that represent self-healing or transient issues. Yujun et al. [22] study the problem of grouping different incidents with intrinsic relationships. Incident detection or prediction has gained increasing interest in recent years. A few recent work leverages customer feedback [55], tweets [15], or alerts with rich textual information [54] to detect real-time issues for online systems. They focus on extracting effective indicators from textual information. However, in our system, the alerts carry only machine generated texts as shown in Fig. 9 which is different from natural languages. Our previous work AirAlert [23] uses monitor generated alerts to predict several specific types of outages. In this work, Warden is designed for generic incident detection. We compare the performance of Warden with AirAlert in Sec. 4.3.

7 Conclusion and Future Work

In this work, we propose Warden, a framework to automatically detect incidents. We train an inference model based on historic failure patterns with input from a set of carefully selected monitors. Upon detecting potential incidents, Warden extracts a set of related alert signals and notifies relevant on-call engineers. This information assists the on-call engineers to prioritize their tasks and initiate cross-team collaboration. We have evaluated Warden using data collected from 26 major services on Azure. The evaluation results confirm the effectiveness of Warden. Furthermore, Warden has been successfully deployed in production.

We notice that not all incidents are covered by the monitoring system. Therefore, in our future work, we plan to exploit more signals such as the customer submitted support cases and social streams for incident detection.

Acknowledgement

We thank Kiran-Kumar Muniswamy-Reddy, our shepherd, and the anonymous reviewers for their tremendous feedback and comments. We also thank our partners Januelle Roldan, Sai Ramani, Navendu Jain, Irfan Mohiuddin, Andrew Edwards Douglas Phillips, Rakesh Namineni, David Wilson, Paul Lorimer, Victor Rühle, Jim Kleewein, Perry Clarke, Alex Dong, Wesley Miao, Andrew Zhou, Robert Gu, Yining Wei, Yingnong Dang, and Murali Chintalapati in the production teams for their valuable feedback on this work and using our system. Hongyu Zhang is supported by Australian Research Council (ARC) Discovery Project DP200102940.

References

- [1] Data incident response process. https://services.google.com/fh/files/misc/data_incident_response_2018.pdf, 2018. [Online; accessed 10-August-2020].
- [2] Site reliability engineering documentation. <https://docs.microsoft.com/en-us/azure/site-reliability-engineering/>, 2018. [Online; accessed 10-August-2020].
- [3] Atlassian Incident Handbook. <https://www.atlassian.com/incident-management/handbook/incident-response>, 2020. [Online; accessed 10-August-2020].
- [4] Bringing order to chaos: The role of the incident commander. <https://www.atlassian.com/incident-management/incident-response/incident-commander>, 2020. [Online; accessed 10-August-2020].
- [5] Computer Security Incident Handling Guide. <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-61r2.pdf>, 2020. [Online; accessed 10-August-2020].
- [6] Managing Incidents. <https://landing.google.com/sre/sre-book/chapters/managing-incidents/>, 2020. [Online; accessed 10-August-2020].
- [7] Amazon CloudWatch. <https://aws.amazon.com/cloudwatch/>, 2021. [Online; accessed 28-Apr-2021].
- [8] AWS Post-Event Summaries. <https://aws.amazon.com/cn/premiumsupport/technology/pes/>, 2021. [Online; accessed 28-Apr-2021].
- [9] Azure Cognitive Services. <https://azure.microsoft.com/en-us/services/cognitive-services/>, 2021. [Online; accessed 28-Apr-2021].
- [10] Azure status history. <https://status.azure.com/en-us/status/history/>, 2021. [Online; accessed 28-Apr-2021].
- [11] Google Cloud Status Dashboard. <https://status.cloud.google.com/summary>, 2021. [Online; accessed 28-Apr-2021].
- [12] Splunk: Machine Learning and Anomaly Detection. <https://www.splunk.com/>, 2021. [Online; accessed 28-Apr-2021].
- [13] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *NSDI*, 2018.
- [14] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the blame game out of data centers operations with netpilot. In *SIGCOMM*, 2016.

- [15] Eriq Augustine, Cailin Cushing, Alex Dekhtyar, Kevin McEntee, Kimberly Paterson, and Matt Tognetti. Outage detection via real-time social stream analysis: Leveraging the power of online complaints. In *WWW*, 2012.
- [16] Ranjita Bhagwan, Rahul Kumar, Ramachandran Ramjee, George Varghese, Surjyakanta Mohapatra, Hemanth Manoharan, and Piyush Shah. Adtributor: Revenue debugging in advertising systems. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.
- [17] Junjie Chen, Xiaoting He, Qingwei Lin, Yong Xu, Hongyu Zhang, Dan Hao, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. An empirical investigation of incident triage for online service systems. In *ICSE-SEIP*, 2019.
- [18] Junjie Chen, Xiaoting He, Qingwei Lin, Hongyu Zhang, Dan Hao, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. Continuous incident triage for large-scale online service systems. In *ASE*, pages 364–375, 11 2019.
- [19] Junjie Chen, Shu Zhang, Xiaoting He, Qingwei Lin, Hongyu Zhang, Dan Hao, Yu Kang, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. How incidental are the incidents? characterizing and prioritizing incidents for large-scale online service systems. In *ASE*, 2020.
- [20] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *KDD*, 2016.
- [21] Yujun Chen, Xian Yang, Hang Dong, Xiaoting He, Hongyu Zhang, Qingwei Lin, Junjie Chen, Pu Zhao, Yu Kang, Feng Gao, Zhangwei Xu, and Dongmei Zhang. Identifying linked incidents in large-scale online service systems. In *ESEC/FSE*, 2020.
- [22] Yujun Chen, Xian Yang, Hang Dong, Xiaoting He, Hongyu Zhang, Qingwei Lin, Junjie Chen, Pu Zhao, Yu Kang, Feng Gao, Zhangwei Xu, and Dongmei Zhang. Identifying linked incidents in large-scale online service systems. In *ESEC/FSE*, 2020.
- [23] Yujun Chen, Xian Yang, Qingwei Lin, Dongmei Zhang, Hang Dong, Yong Xu, Hao Li, Yu Kang, Hongyu Zhang, Feng Gao, Zhangwei Xu, and Yingnong Dang. Outage prediction and diagnosis for cloud service systems. *WWW*, pages 2659–2665, 2019.
- [24] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.
- [25] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *ASPLOS*, 2019.
- [26] Silviu Guiaşu. *Information theory with new applications*. McGraw-Hill Companies, 1977.
- [27] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why does the cloud stop computing? Lessons from hundreds of service outages. *SoCC*, 2016.
- [28] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliver, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Deepthi Srinivasan, Biswaranjan Panda, Andrew Baptist, 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 *FAST*, volume 14, 2018.
- [29] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi Wei Lin, and Varugis Kurien. Pingmesh: A Large-scale system for data center network latency measurement and analysis. *SIGCOMM*, pages 139–152, 2015.
- [30] Hao Hu, Hongyu Zhang, Jifeng Xuan, and Weigang Sun. Effective bug triage based on historical bug-fix information. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 122–132, 2014.
- [31] 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. *HotOS*, Part F1293:150–155, 2017.
- [32] Peng Huang, Jacob R Lorch, Lidong Zhou, Chuanxiong Guo, and Yingnong Dang. Capturing and Enhancing In Situ System Observability for Failure Detection Capturing and Enhancing In Situ System Observabil. *OSDI*, 2018.
- [33] Paul Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bull Soc Vaudoise Sci Nat*, 37:547–579, 1901.
- [34] Nader Jafari Rad and Sayyed Heidar Jafari. Passive Realtime Datacenter Fault Detection and Localization. In *NSDI*, 2017.
- [35] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. Performance monitoring and root cause analysis for cloud-hosted web applications. In *Proceedings of the 26th International Conference on World Wide Web*, pages 469–478, 2017.

- [36] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *NIPS*. 2017.
- [37] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research*, 2017.
- [38] Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, and Dongmei Zhang. idice: problem identification for emerging issues. In *ICSE*, pages 214–224, 2016.
- [39] Jian-Guang Lou, Qingwei Lin, Rui Ding, Qiang Fu, Dongmei Zhang, and Tao Xie. Experience report on applying software analytics in incident management of online service. *Automated Software Engineering*, 2017.
- [40] Scott M Lundberg, Gabriel G Erion, and Su-In Lee. Consistent individualized feature attribution for tree ensembles. *arXiv preprint arXiv:1802.03888*, 2018.
- [41] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems*, pages 4765–4774, 2017.
- [42] Meng Ma, Jingmin Xu, Yuan Wang, Pengfei Chen, Zonghua Zhang, and Ping Wang. Automap: Diagnose your microservice-based web applications automatically. In *WWW*, 2020.
- [43] Christoph Molnar. *Interpretable Machine Learning*. 2019. <https://christophm.github.io/interpretable-ml-book/>.
- [44] Biswaranjan Panda, Deepthi Srinivasan, Karan Gupta, Vinayak Khot, Huan Ke, and Haryadi S Gunawi. IASO: A Fail-Slow Detection and Mitigation Framework for Distributed Storage Services. *ATC*, 2019.
- [45] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In *SIGKDD*, 2016.
- [46] Sara Silva, Rúben Pereira, and Ricardo Ribeiro. Machine learning in incident categorization automation. In *CISTI*. IEEE, 2018.
- [47] Erik Štrumbelj and Igor Kononenko. Explaining prediction models and individual predictions with feature contributions. *Knowledge and information systems*, 41(3):647–665, 2014.
- [48] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, Dong Xiang, and Implementation Nsdi. NetBouncer: Active Device and Link Failure Localization in Data Center Networks. In *NSDI*, pages 1–14, 2019.
- [49] Sean J Taylor and Benjamin Letham. Forecasting at scale. *The American Statistician*, 2018.
- [50] Cover TM and Thomas JA. Elements of information theory. *Wiley Series in Telecommunications*, 1991.
- [51] Erci Xu, Yikang Xu, Jiesheng Wu, Mai Zheng, Feng Qin, and Alibaba Group. Lessons and Actions: What We Learned from 10K SSD-Related Storage System Failures. *ATC*, 2019.
- [52] Hang Zhao, Yujing Wang, Juanyong Duan, Congrui Huang, Defu Cao, Yunhai Tong, Bixiong Xu, Jing Bai, Jie Tong, and Qi Zhang. Multivariate time-series anomaly detection via graph attention network. In *ICDM*, 2020.
- [53] Nengwen Zhao, Junjie Chen, Xiao Peng, Honglin Wang, Xinya Wu, Yuanzong Zhang, Zikai Chen, Xiangzhong Zheng, Xiaohui Nie, Gang Wang, Yong Wu, Fang Zhou, Wenchi Zhang, Kaixin Sui, and Dan Pei. Understanding and handling alert storm for online service systems. *ICSE-SEIP '20*, 2020.
- [54] Nengwen Zhao, Junjie Chen, Zhou Wang, Xiao Peng, Gang Wang, Yong Wu, Fang Zhou, Zhen Feng, Xiaohui Nie, Wenchi Zhang, Kaixin Sui, and Dan Pei. Real-time incident prediction for online service systems. In *ESEC/FSE*, 2020.
- [55] Wujie Zheng, Haochuan Lu, Yangfan Zhou, Jianming Liang, Haibing Zheng, and Yuetang Deng. ifeedback: Exploiting user feedback for real-time issue detection in large-scale online service systems. In *ASE*, pages 352–363, 11 2019.
- [56] Wubai Zhou, Wei Xue, Ramesh Baral, Qing Wang, Chunqiu Zeng, Tao Li, Jian Xu, Zheng Liu, Larisa Shwartz, and Genady Ya. Grabarnik. Star: A system for ticket analysis and resolution. In *SIGKDD*, 2017.

Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training

Geoffrey X. Yu
University of Toronto
Vector Institute

Yubo Gao
University of Toronto

Pavel Golikov
University of Toronto
Vector Institute

Gennady Pekhimenko
University of Toronto
Vector Institute

Abstract

Deep learning researchers and practitioners usually leverage GPUs to help train their deep neural networks (DNNs) faster. However, choosing *which* GPU to use is challenging both because (i) there are many options, and (ii) users grapple with competing concerns: maximizing compute performance while minimizing costs. In this work, we present a new practical technique to help users make informed and cost-efficient GPU selections: make performance *predictions* with the help of a GPU that the user already has. Our technique exploits the observation that, because DNN training consists of repetitive compute steps, predicting the execution time of a single iteration is usually enough to characterize the performance of an entire training process. We make predictions by scaling the execution time of each operation in a training iteration from one GPU to another using either (i) wave scaling, a technique based on a GPU’s execution model, or (ii) pre-trained multilayer perceptrons. We implement our technique into a Python library called Habitat and find that it makes accurate iteration execution time predictions (with an average error of 11.8%) on ResNet-50, Inception v3, the Transformer, GNMT, and DCGAN across six different GPU architectures. Habitat supports PyTorch, is easy to use, and is open source.¹

1 Introduction

Over the past decade, deep neural networks (DNNs) have seen incredible success across many machine learning tasks [26, 37, 39, 50, 93, 96, 99]—leading them to become widely used throughout academia and industry. However, despite their popularity, DNNs are not always straightforward to use in practice because they can be extremely computationally-expensive to train [23, 53, 95, 109]. This is why, over the past few years, there has been a significant and ongoing effort to bring *hardware acceleration* to DNN training [10, 16, 35, 36, 45, 78, 80].

As a result of this effort, today there is a vast array of hardware options for deep learning users to choose from for

training. These options range from desktop and server-class GPUs (e.g., 2080Ti [70] and A100 [78]) all the way to specialized accelerators such as the TPU [45], AWS Trainium [10], Gaudi [36], IPU [35], and Cerebras WSE [16]. Having all these options offers flexibility to users, but at the same time can also lead to a paradox of choice: which hardware option should a researcher or practitioner use to train their DNNs?

A natural way to start answering this question is to first consider CUDA-enabled GPUs. This is because they (i) are commonly used in deep learning; (ii) are supported by all major deep learning software frameworks (PyTorch [86], TensorFlow [1], and MXNet [19]); (iii) have mature tooling support (e.g., CUPTI [76]); and (iv) are readily available for rent *and* purchase. In particular, when considering GPUs, we find that there are many situations where a deep learning user needs to *choose* a specific GPU to use for training:

- **Choosing between different hardware tiers.** In both academia and industry, deep learning users often have access to several *tiers* of hardware: (i) a workstation with a GPU used for development (e.g., 2080Ti), (ii) a private GPU cluster that is shared within their organization (e.g., RTX6000 [84]), and (iii) GPUs that they can rent in the cloud (e.g., V100 [66]). Each tier offers a different *cost*, *availability*, and *performance* trade-off. For example, a private cluster might be “free” (in monetary cost) to use, but jobs may be queued because the cluster is also shared among other users. In contrast, cloud GPUs can be rented on-demand for exclusive use.
- **Deciding on which GPU to rent or purchase.** Cloud providers make many different GPUs available for rent (e.g., P100 [62], V100, T4 [71], and A100 [78]), each with different performance at different prices. Similarly, a wide variety of GPUs are available for purchase (e.g., 2080Ti, 3090 [82]) both individually and as a part of pre-built workstations [52]. These GPUs can vary up to $6\times$ in price [98] and $6\times$ in peak performance [79].
- **Determining how to schedule a job in a heterogeneous**

¹Habitat is available on GitHub: github.com/geoffxy/habitat [105, 106]

GPU cluster. A compute cluster (e.g., operated by a cloud provider [8,32,58]) may have multiple types of GPUs available that can handle a training workload. Deciding which GPU to use for a job will typically depend on the job’s priority and performance on the GPU being considered [18,61].

- **Selecting alternative GPUs.** When a desired GPU is unavailable (e.g., due to capacity constraints in the cloud), a user may want to select a different GPU with a comparable cost-normalized performance. For example, when training ResNet-50 [37] on Google Cloud [31], we find that both the P100 and V100 have similar cost-normalized throughputs (differing by just 0.8%). If the V100 were to be unavailable,² a user may decide to use the P100 instead since the total training cost would be similar.

What makes these situations interesting is that there is not necessarily a *single* “correct” choice. Users make GPU selections based on whether the performance benefits of the chosen configuration are *worth* the cost to train their DNNs. But making these selections in an informed way is not easy, as performance depends on many factors simultaneously: (i) the DNN being considered, (ii) the GPU being used, and (iii) the underlying software libraries used during training (e.g., cuDNN [74], cuBLAS [77]).

To do this performance analysis today, the common wisdom is to either (i) directly measure the computational performance (e.g., throughput) by actually running the training job on the GPU, or (ii) consult existing benchmarks (e.g., MLPerf [53]) published by the community to get a “ballpark estimate.” While convenient, these approaches also have their own limitations. Making measurements requires users to already have access to the GPUs they are considering; this may not be the case if a user is deciding whether or not to *buy* or *rent* that GPU in the first place. Secondly, benchmarks are usually only available for a subset of GPUs (e.g., the V100 and T4) and only for common “benchmark” models (e.g., ResNet-50 [37] and the Transformer [99]). They are not as helpful if you need an accurate estimate of the performance of a *custom* DNN on a specific GPU (a common scenario when doing deep learning research).

In this work, we make the case for a third complementary approach: making performance *predictions*. Although predicting the performance of general compute workloads can be prohibitively difficult due to the large number of possible program phases, we observe that DNN training workloads are special because they contain *repetitive computation*. DNN training consists of repetitions of the same (relatively short) training iteration, which means that the performance of an entire training process can be characterized by just a few training iterations.

We leverage this observation to build a new technique that *predicts* a DNN’s training iteration execution time for a given

²In our experience, we often ran into situations where the V100 was unavailable for rent because the cloud provider had an insufficient supply.

```
import habitat

tracker = habitat.OperationTracker(
    origin_device=habitat.Device.RTX2070,
)

with tracker.track():
    run_my_training_iteration()

trace = tracker.get_tracked_trace()
print("Pred. iter. exec. time: {:.2f} ms".format(
    trace.to_device(habitat.Device.V100).run_time_ms,
))
```

Listing 1: An example of how Habitat can be used to make iteration execution time predictions.

batch size and GPU using both *runtime information* and *hardware characteristics*. We make predictions in two steps: (i) we measure the execution time of a training iteration on an *existing* GPU, and then (ii) we scale the measured execution times of each individual operation onto a *different* GPU using either wave scaling or pre-trained multilayer perceptrons (MLPs) [29]. Wave scaling is a technique that applies *scaling factors* to the GPU kernels in an operation, based on a mix of the ratios between the two GPUs’ memory bandwidth and compute units. We use MLPs for certain operations (e.g., convolution) where the kernels used differ between the two GPUs; we describe this phenomenon and the MLPs in more detail in Sections 3.2 and 3.4. We believe that using an existing GPU to make operation execution time predictions for a different GPU is reasonable because deep learning users often already have a local GPU that they use for development.

We implement our technique into a Python library that we call Habitat, and evaluate its prediction accuracy on five DNNs that have applications in image classification, machine translation, and image generation: (i) ResNet-50, (ii) Inception v3 [97] (iii) the Transformer, (iv) GNMT [102], and (v) DCGAN [89]. We use Habitat to make iteration execution time predictions across six different GPUs and find that it makes accurate predictions with an average error of 11.8%. Additionally, we present two case studies to show how Habitat can be used to help users make accurate cost-efficient GPU selections according to their needs (Section 5.3).

We designed Habitat to be easy and practical to use (see Listing 1). Habitat currently supports PyTorch [86] and is open source: github.com/geoffxy/habitat [105, 106].

In summary, this work makes the following contributions:

- Wave scaling: a new technique that scales the execution time of a kernel measured on one GPU to a different GPU by using scaled ratios between the (i) number of compute units on each GPU, and (ii) their memory bandwidths.
- The implementation and evaluation of Habitat: a new library that uses wave scaling along with pre-trained MLPs to predict the execution time of DNN training iterations on different GPUs.

2 Why Predict the Computational Training Performance of DNNs on Different GPUs?

This work presents a new practical technique for predicting the execution time of a DNN training iteration on different GPUs, with the goal of helping deep learning users make informed cost-efficient GPU selections. However, a common first question is to ask why we need to make these performance predictions in the first place. Could other performance comparison approaches (e.g., simple heuristics or measurements) be used instead? In this section, after providing some background about DNN training, we outline the problems with these alternative approaches to further motivate the need for practical performance predictions.

2.1 Background on DNN Training

DNNs, at their heart, are mathematical functions that produce predictions given an input and a set of *learned* parameters, also known as *weights* [29]. They are built by combining together a series of different *layers*, each of which may contain weights. The layers map to mathematical operations. For example, a fully connected layer is implemented using matrix multiplication [29]. To produce predictions, a DNN takes a tensor (an n -dimensional array) as input and applies the operations associated with each layer in sequence.

Training. A DNN learns its weights in an *iterative* process called training. Each training iteration operates on a batch of labelled inputs and consists of a *forward* pass, *backward* pass (using backpropagation [90]), and *weight update*. The forward and backward passes compute gradients for the weights, which are then used by an optimization algorithm (e.g., stochastic gradient descent [12] or Adam [49]) to update the weights so that the DNN produces better predictions. These steps are *repeated* until the DNN makes acceptably accurate predictions.

Computational performance. Although conceptually simple, prior work has shown that DNN training can be an extremely time-consuming process [23, 53, 95, 109]. There are two primary factors that influence the time it takes a DNN to reach an acceptable accuracy during training [59]: (i) statistical efficiency, and (ii) hardware efficiency. Statistical efficiency governs the *number* of training iterations (i.e., weight updates) required to reach a target test accuracy whereas hardware efficiency governs how *quickly* a training iteration runs. In this work, we focus on helping deep learning users make informed cost-efficient hardware configuration selections to improve their DNN’s *hardware efficiency*. As a result, we compare the performance of different GPUs when training a DNN using the *time it takes a training iteration to run*. This metric equivalently captures the training throughput for that particular DNN.

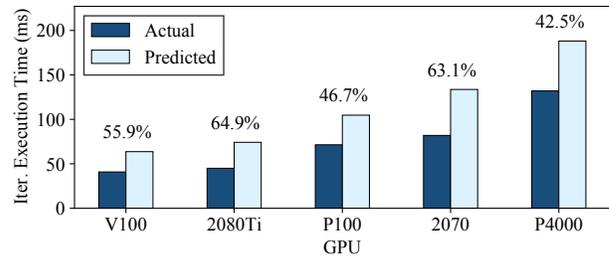


Figure 1: DCGAN iteration execution time predictions, and their errors, made from the T4 using peak FLOPS ratios between the devices. Using simple heuristics can lead to high prediction errors.

2.2 Why Not Measure Performance Directly?

Perhaps the most straightforward approach to compare the performance of different GPUs is to just measure the iteration execution time (and hence, throughput) on each GPU when training a given DNN. However, this approach also has a straightforward downside: it requires the user to actually have access to the GPU(s) being considered in the first place. If a user is looking to buy or rent a cost-efficient GPU, they would ideally want to know its performance on their DNNs *before* spending money to get access to the GPU.

2.3 Why Not Apply Heuristics?

Another approach is to use heuristics based on the hardware specifications published by the manufacturer. For example, one could use the ratio between the peak floating point operations per second (FLOPS) of two GPUs or the ratio between the number of CUDA cores on each GPU. The problem with this approach is that these heuristics do not always work. Heuristics often assume that a DNN training workload can exhaust all the computational resources on a GPU, which is not true in general [109].

To show an example of when simple heuristics do not work well, we use a GPU’s peak FLOPS to make iteration execution time predictions. We measure the execution time of a DCGAN training iteration on the T4³ and then use this measurement to predict the iteration execution time on *different* GPUs by multiplying by the ratio between the devices’ peak FLOPS. Figure 1 shows the measured and predicted execution times on each GPU, along with the prediction error as a percentage. The main takeaway from this figure is that using simple heuristics can lead to high prediction errors; the highest prediction error in this experiment is 64.9%, and all the prediction errors are at least 42.5%. In contrast, Habitat can make these exact same predictions with an average error of 10.2% (maximum 21.8%).

³We use a batch size of 128 LSUN [104] synthetic inputs. See Section 5.1 for details about our methodology.

2.4 Why Not Use Benchmarks?

A third potential approach is to consult published benchmarking results [23,53,81,109]. However, the problem with relying on benchmarking results is that they are limited to a set of “common” DNNs (e.g., ResNet-50 or BERT [26]) and are usually only available for a small selection of GPUs (e.g., the T4, V100, and A100). Moreover, benchmarking results also vary widely among different models and GPUs [53,81,109]. Therefore if no results exist for the GPU(s) a user is considering, or if a user is working with a new DNN architecture, there will be no benchmark results for them to consult.

2.5 Why Not Always Use the “Best” GPU?

Finally, a fourth approach is to always use the most “powerful” GPU available with the assumption that GPUs are already priced based on their performance. Why make performance predictions when the cost-efficiency of popular GPUs should be the same? However, this assumption is a misconception. Prior work has already shown examples where the performance benefits of different GPUs changes depending on the model [18,61,109]. In this work, we also show additional examples in our case studies (Section 5.3) where (i) cost-efficiency leads to selecting a different GPU, and (ii) where the V100 does not offer significant performance benefits over a common desktop-class GPU (the 2080Ti).

Summary. Straightforward approaches that users might consider to make GPU selections all have their own downsides. In particular, existing approaches either require access to the GPUs themselves or are only applicable for common DNNs and GPUs. Therefore there is a need for a complementary approach: making performance predictions—something that we explore in this work.

3 Habitat

Our approach to performance predictions is powered by three key observations. In this section, after describing these observations, we outline the key ideas behind Habitat.

3.1 Key Observations

Observation 1: Repetitive computation. While training a DNN to an acceptable accuracy can take on the order of hours to days [23,53,109], a single training iteration takes on the order of hundreds of *milliseconds*. This observation improves the predictability of DNN training as we can characterize the performance of an entire DNN training session using the performance of a single iteration.

Observation 2: Common building blocks among DNNs. Although DNNs can consist of hundreds of operations, they are built using a relatively *small* set of unique operations. For

example, convolutional neural networks typically comprise convolutional, pooling, fully connected, and batch normalization [42] layers. This observation reduces the problem of predicting the performance of an arbitrary DNN’s training iteration to developing prediction mechanisms for a small set of operations.

Observation 3: Runtime information available. When developing DNNs, users often have a GPU available for use in their workstations. These GPUs are used for development purposes and are not necessarily chosen for the highest performance (e.g., 1080Ti [64], TITAN Xp [68]). However, they can be used to provide valuable runtime information about the GPU kernels that are used to implement a given DNN. In Section 3.3, we describe how we can leverage this runtime information to predict the performance of the GPU kernels on different GPUs (e.g., from a desktop-class GPU such as the 2080Ti [70] to a server-class GPU such as the V100 [66,67]).

3.2 Habitat Overview

Habitat records information at runtime about a DNN training iteration for a specific batch size on a given GPU (*Observation 3*) and then uses that information to predict the training iteration execution time on a *different* GPU (for the same batch size). Predicting the iteration execution time is enough (*Observation 1*) to compute metrics about the entire training *process* on different GPUs. These predicted metrics, such as the training throughput and cost-normalized throughput, are then used by end-users (e.g., deep learning researchers) to make informed hardware selections.

To actually make these predictions for a different GPU, Habitat predicts the new execution time of each individual operation in a training iteration. Habitat then adds these predicted times together to arrive at an execution time prediction for the entire iteration. For an individual operation, Habitat makes predictions using either (i) wave scaling (Section 3.3), or (ii) pre-trained MLPs (Section 3.4).

The reason why Habitat uses two techniques together is that wave scaling assumes that the *same* GPU kernels are used to implement a given DNN operation on each GPU. However, some DNN operations are implemented using *different* GPU kernels on different GPUs (e.g., convolutions, recurrent layers). This is done for performance reasons as these operations are typically implemented using proprietary kernel libraries that leverage GPU architecture-specific kernels (e.g., cuDNN [21], cuBLAS [77]). We refer to these operations as *kernel-varying*, and scale their execution times to different GPUs using pre-trained MLPs. Habitat uses wave scaling for the rest of the operations, which we call *kernel-alike*.

3.3 Wave Scaling

Wave scaling works by scaling the execution times of the *kernels* used to implement a kernel-alike DNN operation. The

computation performed by a GPU kernel is partitioned into groups of threads called *thread blocks* [28], which typically execute in concurrent groups, resulting in *waves* of execution. The key idea behind wave scaling is to compute the number of *thread block waves* in a kernel and scale the wave execution time using *ratios* between the origin and destination GPUs.

We describe wave scaling formally in Equation 1. Let T_i represent the execution time of the kernel on GPU i , B the number of thread blocks in the kernel, W_i the number of thread blocks in a wave on GPU i , D_i the memory bandwidth on GPU i , and C_i the clock frequency on GPU i . Here we let $i \in \{o, d\}$ to represent the origin and destination GPUs. By measuring T_o (*Observation 3*), wave scaling predicts T_d using

$$T_d = \left\lceil \frac{B}{W_d} \right\rceil \left(\frac{D_o W_d}{D_d W_o} \right)^\gamma \left(\frac{C_o}{C_d} \right)^{1-\gamma} \left\lceil \frac{B}{W_o} \right\rceil^{-1} T_o \quad (1)$$

where $\gamma \in [0, 1]$ represents the “memory bandwidth bound-ness” of the kernel. Habitat selects γ by measuring the kernel’s arithmetic intensity and then leveraging the roofline model [101] (see Section 4.2).

As shown in Equation 1, wave scaling uses the ratios between the GPUs’ (i) memory bandwidths, (ii) clock frequencies, and (iii) the size of a wave on each GPU. The intuition behind factors (i) and (iii) is that a higher relative memory bandwidth allows more memory requests to be served in parallel whereas having more thread blocks in a wave results in more memory requests being made. Thus, everything else held constant, waves in memory bandwidth bound kernels (i.e., large γ) should see speedups on GPUs with more memory bandwidth. The intuition behind factor (ii) is that higher clock frequencies may benefit waves in compute bound kernels (i.e., small γ).⁴

For large $\lceil B/W_i \rceil$ (i.e., when there are a large number of waves) we get that $\lceil B/W_i \rceil \approx B/W_i$. In this case, Equation 1 simplifies to

$$T_d = \left(\frac{D_o}{D_d} \right)^\gamma \left(\frac{W_o}{W_d} \right)^{1-\gamma} \left(\frac{C_o}{C_d} \right)^{1-\gamma} T_o \quad (2)$$

Habitat uses Equation 2 to predict kernel execution times because we find that in practice, most kernels are composed of many thread blocks.

Habitat computes W_i for each kernel and GPU using the thread block occupancy calculator that is provided as part of the CUDA Toolkit [80]. We obtain C_i from each GPU’s specifications, and we obtain D_i by measuring the achieved bandwidth on each GPU ahead of time. Note that we make these measurements once and then distribute them in a configuration file with Habitat.

⁴The clock’s impact on execution time depends on other factors too (e.g., the GPU’s instruction set architecture). Wave scaling aims to be a simple and understandable model and therefore does not model these complex effects.

3.4 MLP Predictors

To handle kernel-varying operations, Habitat uses pre-trained MLPs to make execution time predictions. We treat this prediction task as a regression problem: given a series of input features about the operation and a target GPU (described below), predict the operation’s execution time on that target GPU. We learn an MLP for each kernel-varying operation that Habitat currently supports: (i) convolutions (2-dimensional), (ii) LSTMs [38], (iii) batched matrix multiplies, and (iv) linear layers (matrix multiply with an optional bias term). As we show in Section 5, relatively few DNN operations are kernel-varying⁵ and so training separate MLPs for each of these operations is a feasible approach. Furthermore, these MLPs can be used for many different DNNs as these operations are common “building blocks” used in DNNs (*Observation 2*).

Input features. Each operation-specific MLP takes as input: (i) layer dimensions (e.g., the number of input and output channels in a convolution); (ii) the memory capacity and bandwidth on the target GPU; (iii) the number of streaming multiprocessors (SMs) on the target GPU; and (iv) the peak FLOPS of the target GPU, specified by the manufacturer.

Model architecture. Each MLP comprises an input layer, eight hidden layers, and an output layer that produces a single real number—the predicted execution time (this includes the forward and backward pass) for the MLP’s associated operation. We use ReLU activation functions in each layer and we use 1024 units in each hidden layer. We outline the details behind our datasets and how these MLPs are trained in Section 4.3.

4 Implementation Details

Habitat is built to work with PyTorch [86]. However, the ideas behind Habitat are general and straightforward to implement in other frameworks as well. Habitat performs its analysis using a DNN’s computation graph, which is also available in other frameworks (e.g., TensorFlow [1] and MXNet [19]).

4.1 Extracting Runtime Metadata

Habitat extracts runtime metadata in a training iteration by “monkey patching” PyTorch operations with special wrappers. These wrappers allow Habitat to intercept and keep track of all the operations that run in one training iteration, as they are executed. As shown in Listing 1, users explicitly indicate to Habitat when to start and stop tracking the operations in a DNN by calling `track()`.

Execution time. To measure the execution time of each operation, Habitat re-runs each operation independently with

⁵This is, in part, because implementing performant architecture-specific kernels for each kernel-varying operation takes significant engineering effort.

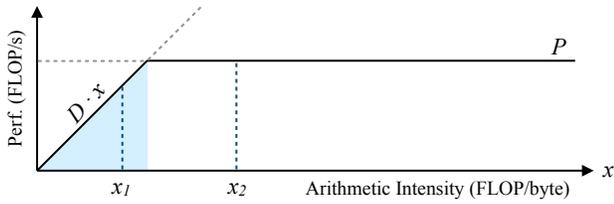


Figure 2: An example roofline model. If a kernel’s arithmetic intensity falls in the shaded region, it is considered memory bandwidth bound (x_1); otherwise, it is considered compute bound (x_2).

the same inputs as recorded when the operation was intercepted. Habitat also measures the execution time associated with the operation’s backward pass, if applicable. The reason why Habitat makes measurements by re-running the individual operations is that the operations could be very short (in execution time). Thus, Habitat needs to run them multiple times to make accurate measurements. Habitat uses CUDA events [73] to make these timing measurements.

Kernel metadata and metrics. Habitat uses CUPTI [76] to record execution times and metrics (see Section 4.2) for the kernels used to implement each operation in the DNN. This information is used by wave scaling.

4.2 Selecting Gamma (γ)

Recall from Section 3.3 that wave scaling scales its ratios using γ , a factor that represents the “memory bandwidth boundedness” of a kernel. In this section, we describe in more detail how Habitat automatically selects γ for each kernel.

Roofline model. Habitat uses the roofline model [101] to estimate a kernel’s memory boundedness. Figure 2 shows an example roofline model. The roofline model introduces the notion of a kernel’s arithmetic intensity: the number of floating point operations it performs per byte of data read or written to memory (represented by x in Figure 2).

A key idea behind the roofline model is that it models a kernel’s peak performance as the minimum of either the hardware’s peak performance (P) or the hardware’s memory bandwidth times the kernel’s arithmetic intensity ($D \cdot x$) [101]. This minimum is shown by the solid line in Figure 2. The arithmetic intensity where these two limits meet is called the “ridge point” (R), where $R = P/D$. The model considers a kernel with an arithmetic intensity of x to be memory bandwidth bound if $x < R$ and compute bound otherwise. For example, in Figure 2, a kernel with an arithmetic intensity of x_1 would be considered memory bandwidth bound whereas a kernel with an intensity of x_2 would be considered compute bound.

Wave scaling leverages the observation that a kernel’s arithmetic intensity is fixed across GPUs (i.e., arithmetic intensity only depends on the kernel’s code). R changes across GPUs

because P and D vary among GPUs, but can be computed using a GPU’s performance specifications. Therefore, if Habitat computes a kernel’s arithmetic intensity, it can use the arithmetic intensity’s distance from the destination GPU’s ridge point to estimate the kernel’s memory bandwidth boundedness (on the destination GPU).

Selecting γ . When profiling each kernel, Habitat gathers metrics that allow it to empirically calculate the kernel’s arithmetic intensity (floating point efficiency, number of bytes read and written to DRAM). If we let x be the kernel’s measured arithmetic intensity and $R = P/D$ for the destination GPU (using the notation presented above), Habitat sets γ using

$$\gamma = \begin{cases} (-0.5/R)x + 1 & \text{if } x < R \\ 0.5R/x & \text{otherwise} \end{cases} \quad (3)$$

This equation means that γ decreases linearly from 1 to 0.5 as x increases toward R . After passing R , γ approaches 0 as x approaches infinity.

Practical optimizations. In practice, gathering metrics on GPUs is a slow process because the kernels need to be replayed multiple times to capture all the needed performance counters. To address this challenge, we make two optimizations: (i) we cache measured metrics, keyed by the kernel’s name and its launch configuration (number of thread blocks and block size); and (ii) we only measure metrics for operations that contribute significantly to the training iteration’s execution time (e.g., with execution times at or above the 99.5th percentile). Consequently, when metrics are unavailable for a particular kernel, we set $\gamma = 1$. We believe that this is a reasonable approximation because kernel-alike operations tend to be very simple (e.g., element-wise operations) and are therefore usually memory bandwidth bound.

4.3 MLPs: Data and Training

In this section, we describe the details behind Habitat’s MLPs: how we (i) collect training data, (ii) preprocess the data, and (iii) train the MLPs.

4.3.1 Data Collection

We gather training data by measuring the forward and backward pass execution times of kernel-varying operations at randomly sampled *input configurations*. An input configuration is a setting of an operation’s parameters (e.g., batch size and number of channels in a convolution). We use predefined ranges for each operation’s parameters, as described in more detail below, and ignore any configurations that result in running out of memory. We make these measurements for all six of the GPUs listed in Section 5.1. We use the same seed when sampling on different GPUs to ensure we have measurements for the same random input configurations across all the GPUs.

Table 1: A summary of the datasets used for our MLPs.

Operation	Features	Dataset Size
2D Convolution	7+4	91,138 × 6
LSTM	7+4	124,176 × 6
Batched Matrix Multiply	4+4	131,022 × 6
Linear Layer	4+4	155,596 × 6

We create the final dataset by joining data entries that have the same operation and configuration, but with different GPUs.

2D convolutions. For convolutions, we vary the (i) batch size (1 – 64), (ii) number of input (3 – 2048) and output channels (16 – 2048), (iii) kernel size (1 – 11), (iv) padding (0 – 3), (v) stride (1 – 4), (vi) image size (1 – 256), and (vii) whether or not there is a bias weight. We only sample configurations with square images and kernel sizes. During sampling, we ignore any configurations that result in invalid arguments (e.g., a kernel size larger than the image). We selected these parameter ranges by surveying the convolutional neural networks included in PyTorch’s torchvision package [24].

LSTMs. For LSTMs, we vary the (i) batch size (1 – 128), (ii) number of input features (1 – 1280), (iii) number of hidden features (1 – 1280), (iv) sequence length (1 – 64), (v) number of stacked layers (1 – 6), (vi) whether or not the LSTM is bidirectional, and (vii) whether or not there is a bias weight.

Batched matrix multiply (bmm). For a batched matrix multiply of $A \times B$ where $A \in \mathbb{R}^{n \times l \times m}$ and $B \in \mathbb{R}^{n \times m \times r}$, we vary the (i) batch size (n) (1 – 128), and (ii) the l , m , and r dimensions (1 – 1024).

Linear layers. For linear layers, we vary the (i) batch size (1 – 3500), (ii) input features (1 – 32768), (iii) output features (1 – 32768), and (iv) whether or not there is a bias weight.

4.3.2 Data Preprocessing

After collecting data on the GPUs, we build one dataset per operation by (i) adding the forward and backward execution times to arrive at a single execution time for each operation instance on a particular GPU, and (ii) attaching additional GPU hardware features to each of these data points. We attach the GPU’s (i) memory capacity and bandwidth; (ii) number of streaming multiprocessors (SMs); and (iii) peak FLOPS, as specified by the GPU manufacturer.

We present the characteristics of the final datasets in Table 1. We add four to the number of features to account for the four GPU features (described above) that we add to each data point. Similarly, in the dataset size column we show the total number of unique operation configurations that we sample. We multiply by six because we make measurements on six different GPUs.

Table 2: The GPUs we use in our evaluation.

GPU	Generation	Mem.	Mem. Type	SMs	Rental Cost ⁶
P4000 [65]	Pascal [63]	8 GB	GDDR5 [56]	14	–
P100 [62]		16 GB	HBM2 [4]	56	\$1.46/hr
V100 [66]	Volta [67]	16 GB	HBM2	80	\$2.48/hr
2070 [69]	Turing [72]	8 GB	GDDR6 [57]	36	–
2080Ti [70]		11 GB	GDDR6	68	–
T4 [71]		16 GB	GDDR6	40	\$0.35/hr

4.3.3 Training

We implement our MLPs using PyTorch. We train each MLP for 80 epochs using the Adam optimizer [49] with a learning rate of 5×10^{-4} , weight decay of 10^{-4} , and a batch size of 512 samples. We reduce the learning rate to 10^{-4} after 40 epochs. We use the mean absolute percentage error as our loss function:

$$L = \frac{1}{n} \sum_{i=1}^n \left| \frac{\text{predicted}_i - \text{measured}_i}{\text{measured}_i} \right|$$

We assign 80% of our data samples to the training set and the rest to our test set. None of the configurations that we test on in Section 5 appear in our training sets. We normalize the inputs by subtracting by the mean and dividing by the standard deviation of the input features in our training set.

5 Evaluation

Habitat is meant to be used by deep learning researchers and practitioners to predict the *potential compute performance* of a given GPU so that they can make *informed* cost-efficient choices when selecting GPUs for training. Consequently, in our evaluation our goals are to determine (i) how *accurately* Habitat can predict the training iteration execution time on GPUs with different architectures, and (ii) whether Habitat can correctly predict the relative cost-efficiency of different GPUs when used to train a given model. Overall, we find that Habitat makes iteration execution time predictions across pairs of *six* different GPUs with an average error of 11.8% on ResNet-50 [37], Inception v3 [97], the Transformer [99], GNMT [102], and DCGAN [89].

5.1 Methodology

Hardware. In our experiments, we use the GPUs listed in Table 2. For the P4000, 2070, and 2080Ti we use machines whose configurations are listed in Table 3. For the T4 and V100, we use g4dn.xlarge and p3.2xlarge instances on AWS respectively [7]. For the P100, we use Google Cloud’s

⁶Google Cloud pricing in us-central1, as of June 2021.

Table 3: The machines we use in our evaluation.

CPU	Freq.	Cores	Main Mem.	GPU
Xeon E5-2680 v4 [41]	2.4 GHz	14	128 GB	P4000
Ryzen TR 1950X [5]	3.4 GHz	16	16 GB	2070
EPYC 7371 [6]	3.1 GHz	16	128 GB	2080Ti

Table 4: The DNNs and training configurations we use.

Application	Model	Arch. Type	Dataset
Image Classif.	ResNet-50 [37]	Convolution	ImageNet [91]
	Inception v3 [97]		
Machine Transl.	GNMT [102]	Recurrent	WMT’16 [11] (EN-DE)
	Transformer [99]	Attention	
Image Gen.	DCGAN [89]	Convolution	LSUN [104]

n1-standard instances [30] with 4 vCPUs and 15 GB of system memory.

Runtime environment. We run our experiments inside Docker containers [27]. Our container image uses Ubuntu 18.04 [15], CUDA 10.1 [80], and cuDNN 7 [74]. On cloud instances, we use the NVIDIA GPU Cloud Image, version 20.06.3 [83]. We use PyTorch 1.4.0 [86] for all experiments.

Models and datasets. We evaluate Habitat by predicting the training iteration execution time for the models listed in Table 4 on different GPUs. For ResNet-50 and Inception v3 we use stochastic gradient descent [12]. We use Adam [49] for the rest of the models. We use synthetic data (sampled from a normal distribution) of the *same size* as samples from each dataset.⁷ For the machine translation models, we use a fixed sequence length of 50—the longest sentence length typically used—to show how Habitat can make predictions for a lower bound on the computational performance.

Metrics. In our experiments, we measure and predict the *training iteration execution time*—the wall clock time it takes to perform one training step on a batch of inputs. We use the training iteration execution time to compute the training *throughput* and *cost-normalized throughput* for our analysis. The training throughput is the batch size divided by the iteration execution time. The cost-normalized throughput is the throughput divided by the hourly cost of renting the hardware.

Measurements. We use CUDA events to measure the execution time of training iterations and DNN operations. We run 3 warm up repetitions, which we discard, and then record the average execution time over 3 further repetitions. We use CUPTI [76] to measure a kernel’s execution time.

⁷We verified that the training computation time does not depend on the values of the data itself.

5.2 How Accurate are Habitat’s Predictions?

To evaluate Habitat’s prediction accuracy, we use it to make training iteration execution time predictions for ResNet-50, Inception v3, the Transformer, GNMT, and DCGAN on all six GPUs listed in Section 5.1. Recall that Habitat makes execution time predictions by *scaling* the execution time of a model and specific batch size measured on one GPU (the “origin” GPU) to another (the “destination” GPU). As a result, we use all 30 possible (origin, destination) pairs of these six GPUs in our evaluation.

5.2.1 End-to-End Prediction Accuracy

Figure 3 shows Habitat’s prediction errors for these aforementioned end-to-end predictions. Each subfigure shows the predictions for all five models on a specific destination GPU. We make predictions for three different batch sizes (shown on the figures) and plot both the predicted and measured iteration execution times. Since we consider all possible pairs of our six GPUs, for each destination GPU we plot the average predicted execution times among the five origin GPUs. Similarly, we show the average prediction error above each bar. From these figures, we can draw three major conclusions.

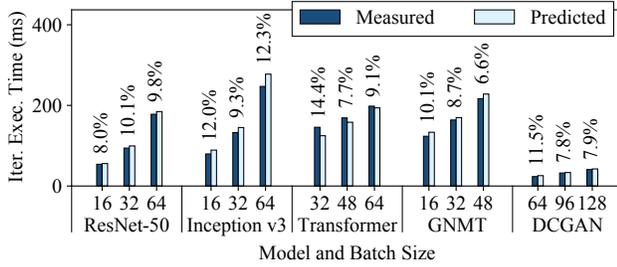
First, Habitat makes accurate end-to-end iteration execution time predictions since the average prediction error across all GPUs and models is 11.8%. The average prediction error across all ResNet-50, Inception v3, Transformer, GNMT, and DCGAN configurations are 13.4%, 9.5%, 12.6%, 11.2%, and 12.3% respectively.

Second, Habitat can predict the iteration execution time across GPU *generations*, which have different architectures, and across *classes* of GPUs. The GPUs we use span three generations (Pascal [63], Volta [67], and Turing [72]) and include desktop, professional workstation, and server-class GPUs.

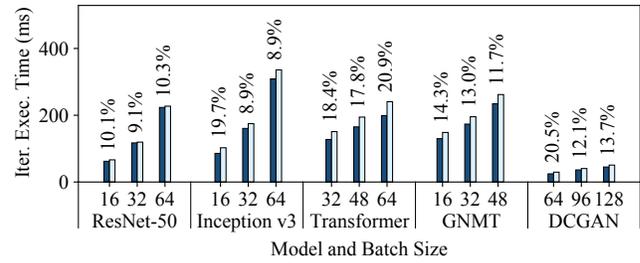
Third, Habitat is *general* since it supports different types of DNN architectures. Habitat works with convolutional neural networks (e.g., ResNet-50, Inception v3, DCGAN), recurrent neural networks (e.g., GNMT), and other neural network architectures such as the attention-based Transformer. In particular, Habitat makes accurate predictions for ResNet, Inception, and DCGAN despite the significant differences in their architectures; ResNet has a “straight-line” computational graph, Inception has a large “fanout” in its graph, and DCGAN is a generative-adversarial model.

5.2.2 Prediction Error Breakdown

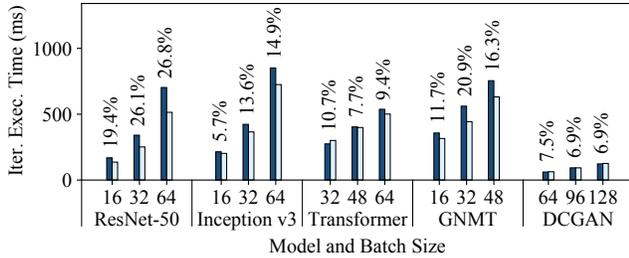
Figure 4 shows a breakdown of the prediction errors for the execution time of individual operations, which are listed on the *x*-axis. The operations predicted using the MLP predictors are shown on the left (conv2d, lstm, bmm, and linear). Wave scaling is used to predict the rest of the operations. Above each bar, we also show the *importance* of each operation as



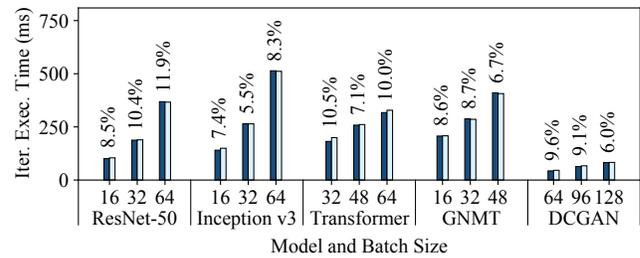
(a) Predictions onto the V100



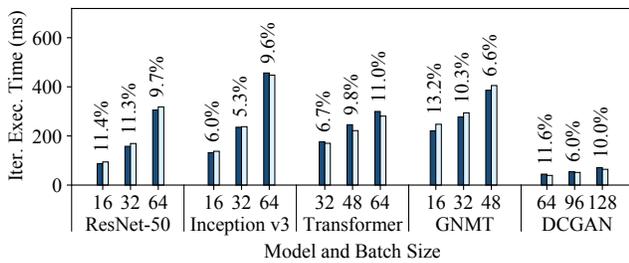
(b) Predictions onto the 2080Ti



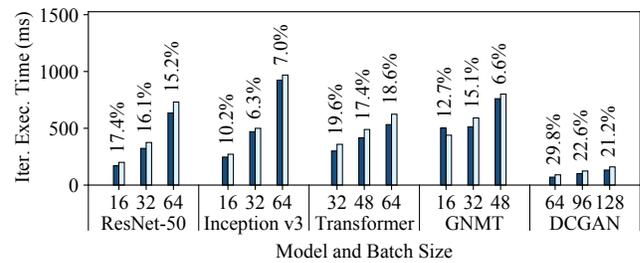
(c) Predictions onto the T4



(d) Predictions onto the 2070



(e) Predictions onto the P100



(f) Predictions onto the P4000

Figure 3: Iteration execution time predictions averaged across all other “origin” GPUs we evaluate.

a percentage of the iteration execution time, averaged across all five DNNs. The prediction errors are averaged among all pairs of the six GPUs that we evaluate and among ResNet-50, Inception v3, the Transformer, GNMT, and DCGAN. From this figure, we can draw two major conclusions.

First, MLP predictors can be used to make accurate predictions for kernel-varying operations as the average error among all the conv2d, lstm, bmm, and linear operations is 18.0%. Second, wave scaling can make accurate predictions for important operations; the average error for wave scaling predictions is 29.8%. Although wave scaling’s predictions for some operations (e.g., `__add__`, `scatter`) have high errors, these operations do not make up a significant proportion of the training iteration execution time (having an overall importance of at most 0.3%).

5.2.3 Prediction Contribution Breakdown

We also examine how wave scaling and the MLPs each contribute to making Habitat’s end-to-end predictions. In our

evaluation, Habitat uses wave scaling for 95% of the unique operations; it uses MLPs for the other 5%. In contrast, when looking at execution time, Habitat uses wave scaling to predict 46% of an iteration’s execution time on average; it uses MLPs for the other 54%.

These breakdowns show that *both* wave scaling and the MLPs contribute non-trivially to Habitat’s predictions—each is responsible for roughly half of an iteration’s execution time. Additionally, the unique operation breakdown shows that most operations are predicted using wave scaling. This observation highlights a strength of Habitat’s hybrid approach of using both wave scaling and MLPs: most operations can be automatically predicted using wave scaling; MLPs only need to be trained for a few kernel-varying operations.

5.2.4 MLPs: How Many Layers?

In all our MLPs, we use eight hidden layers, each of size 1024. To better understand how the number of layers affects the MLPs’ prediction accuracy, we also conduct a sensitivity

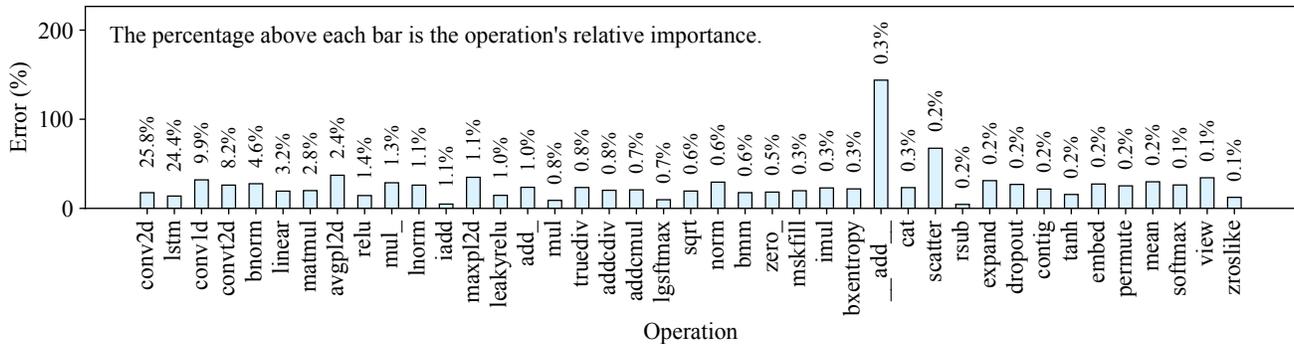


Figure 4: Operation execution time prediction errors, with importance on top of each bar, averaged across all pairs of evaluated GPUs and models. The operation names have been shortened and we only show operations with an importance of at least 0.1%.

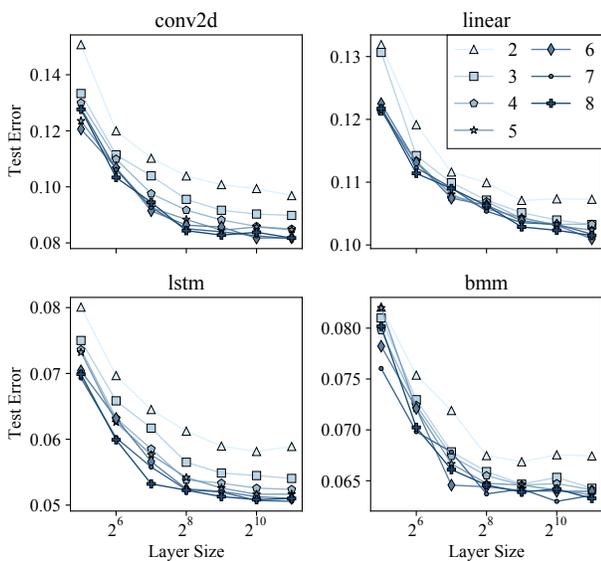


Figure 5: Test error as we vary the number of layers and their sizes in each MLP. The x -axis is in a logarithmic scale.

study where we vary the number of hidden layers in each MLP (2 to 8) along with their size (powers of two: 2^5 to 2^{11}). Figure 5 shows each MLP’s test mean absolute percentage error after being trained for 80 epochs. From this figure we can draw two major conclusions.

First, increasing the number of layers and their sizes leads to lower test errors. Increasing the size of each layer beyond 2^9 seems to lead to diminishing returns on each operation. Second, the MLPs for all four operations appear to follow a similar test error trend. Based on these results, we can also conclude that using eight hidden layers is a reasonable choice.

5.3 Does Habitat Lead to Correct Decisions?

One of Habitat’s primary use cases is to help deep learning users make *informed* and *cost-efficient* GPU selections. In the following two case studies, we demonstrate how Habitat can

make cost-efficiency predictions that empower users to make correct selections according to their needs.

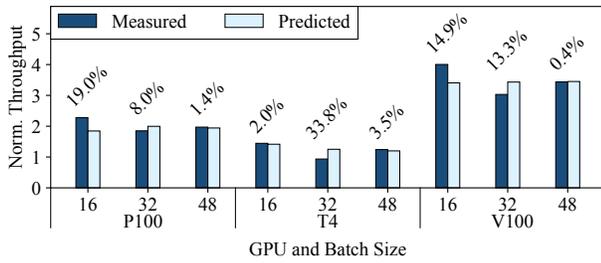
5.3.1 Case Study 1: Should I Rent a Cloud GPU?

As mentioned in Section 1, one scenario a deep learning user may face is deciding whether to rent GPUs in the cloud for training or to stick with a GPU they already have locally (e.g., in their desktop). For example, suppose a user has a P4000 in their workstation and they want to decide whether to rent a P100, T4, or V100 in the cloud to train GNMT.

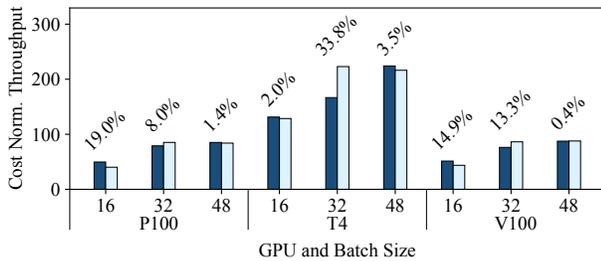
With Habitat, they can use their P4000 to make *predictions* about the computational performance of each cloud GPU to help them make this decision in an informed way. Figure 6a shows Habitat’s throughput predictions for GNMT for the P100, T4, and V100 normalized to the training throughput on the P4000. Additionally, Figure 6b shows Habitat’s predicted training throughputs normalized by each cloud GPU’s rental costs on Google Cloud as shown in Table 2. Note that (i) we make all these predictions with the P4000 as the origin device, (ii) we make our ground truth measurements on Google Cloud instances, and (iii) one can also use Habitat for a similar analysis for other cloud providers. From these results, the user can make two observations.

First, both the P100 and V100 offer training throughput speedups over the P4000 (up to $2.3\times$ and $4.0\times$ respectively) whereas the T4 offers marginal throughput speedups (up to $1.4\times$). However, second, the user would also discover that the T4 is more *cost-efficient* to rent when compared to the P100 and V100 as it has a higher cost-normalized throughput. Therefore, if the user wanted to optimize for maximum computational performance, they would likely choose the V100. But if they were not critically constrained by time and wanted to optimize for cost, sticking with the P4000 or renting a T4 would be a better choice.

Habitat makes these predictions accurately, with an average error of 10.7%. We also note that despite any prediction errors, Habitat still *correctly* predicts the relative ordering of these three GPUs in terms of their throughput and cost-normalized



(a) GNMT training throughput normalized to the P4000



(b) GNMT cost normalized throughput

Figure 6: Habitat’s GNMT training throughput predictions for cloud GPUs, made using a P4000. The percentage error is shown above each prediction.

throughput. For example, in Figure 6b, Habitat correctly predicts that the T4 offers the best cost-normalized throughput on all three batch sizes. These predictions therefore allow users to make correct decisions based on their needs (optimizing for cost or pure performance).

5.3.2 Case Study 2: Is the V100 Always Better?

In the previous case study, Habitat correctly predicts that the V100 provides the best performance despite not being the most cost-efficient to rent. This conclusion may lead a naïve user to believe that the V100 always provides better training throughput over other GPUs, given that it is the most advanced and expensive GPU available in the cloud to rent.⁸ In this case study, we show how Habitat can help a user recognize when the V100 does not offer significant performance benefits for their model.

Suppose a user wants to train DCGAN and already has a 2080Ti that they can use. They want to find out if they should use a different GPU to get better computational performance (training throughput). They can use Habitat to predict the training throughput on other GPUs. Figure 7 shows Habitat’s throughput predictions along with the measured throughput, normalized to the 2080Ti’s training throughput. Note that we use a batch size of 64 as it is the default batch size in the

⁸This is true except for the new A100s, which have only recently become publicly available in the cloud.

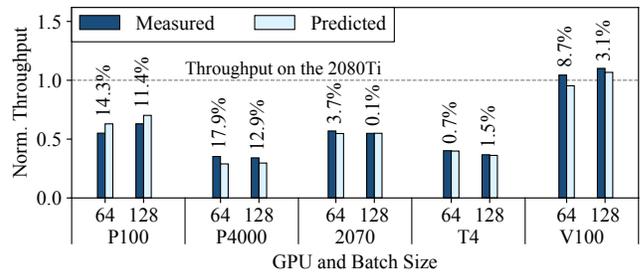


Figure 7: Predicted and measured DCGAN training throughput normalized to the 2080Ti, with prediction errors above each bar. Habitat correctly predicts that the V100’s performance is not significantly better than the 2080Ti.

DCGAN reference implementation [22] and 128 because it is the size reported by the authors in their paper [89].

From this figure, the user would conclude that they should stick to using their 2080Ti as the V100 would not be worth renting. The V100 offers marginal throughput improvements over the 2080Ti (1.1×) while the P100, P4000, 2070, and T4 all do not offer throughput improvements at all. The reason the V100 does not offer any significant benefits over the 2080Ti despite having more computational resources (Table 2) is that DCGAN is a “computationally lighter” model compared to GNMT and so it does not really benefit from a more powerful GPU. Habitat makes these predictions accurately, with an average error of 7.7%.

Summary. These case studies show examples of situations where (i) the GPU offering the highest training throughput is *not* the same as the most cost-efficient GPU, and where (ii) the V100 does *not* offer significantly better performance when compared to a desktop-class GPU (the 2080Ti). Notably, in both case studies, Habitat *correctly* predicts each of these findings. As a result, deep learning researchers and practitioners can rely on Habitat to help them make correct cost-efficient GPU selections according to their needs.

6 Discussion

In this section, we discuss how Habitat can be extended to support additional (i) training setups, and (ii) deep learning frameworks. In doing so, we also highlight opportunities for future work and describe some challenges and opportunities associated with supporting additional hardware accelerators.

6.1 Additional Training Setups

Habitat is designed to make accurate cross-GPU execution time predictions for DNN training. However, users may also face situations where they need performance predictions for more complex training setups such as (i) distributed training [25], (ii) mixed precision training [55], or (iii) needing

predictions for batch sizes larger than what can fit on the origin GPU. In this subsection, we outline how Habitat could be extended to support these setups.

6.1.1 Distributed Training

Predicting the execution time of a distributed training iteration generally reduces to predicting (i) the computation time on the cluster’s GPUs, (ii) the communication time among the GPUs and/or nodes, and (iii) how the communication overlaps with the computation. For data parallel training [25], several prior works present techniques for predicting the data parallel iteration execution time given the execution time on a single GPU [87, 88, 110] (i.e., tasks (ii) and (iii)). Habitat’s computation predictions (task (i)) could be used as an input to these existing techniques.

For more complex distribution schemes such as model parallel [25] and pipeline parallel training [40, 60], Habitat could still be used for task (i), but the user would need to split up their model based on the distributed partitioning scheme before profiling it with Habitat. However, for tasks (ii) and (iii), new prediction techniques would need to be developed. This is something we leave to future work.

6.1.2 Mixed Precision Training

The Daydream paper [110] presents a technique for predicting the performance benefits of switching from single to mixed precision training on the *same* fixed GPU. If users want to know about the performance benefits of mixed precision training on a *different* GPU, they can use Daydream’s technique in conjunction with Habitat.

To show that this combined approach can work in practice, we use a P4000 to predict the execution time of a ResNet-50 mixed precision training iteration on the 2070 and 2080Ti.⁹ On the P4000, we first use Habitat to predict the single precision iteration execution time on the 2070 and 2080Ti. Then, we apply Daydream’s technique to translate these predicted single precision execution times into mixed precision execution times. We also repeat this experiment between the 2070 and 2080Ti. Overall, we find that this approach has an average error of 16.1% for predictions onto the 2070 and 2080Ti.

To distinguish between the errors introduced by Habitat versus Daydream, we also apply Daydream’s technique to the measured (i.e., ground truth) single precision iteration execution times. We find that Daydream’s technique alone has an average error of 10.7% for the 2070 and 2080Ti. Thus we believe the additional error introduced by also using Habitat is reasonable, given the extra functionality. So overall, we conclude that Habitat with Daydream should be able to effectively support mixed precision predictions on other GPUs.

⁹We use the same experimental setup and batch sizes as described and shown in Section 5.1 and Figure 3. We compare our iteration execution time predictions against training iterations performed using PyTorch’s automatic mixed precision module.

6.1.3 Larger Batch Sizes

Recall that Habitat’s iteration execution time predictions are for a model and a specific batch size. This means that the origin GPU must be able to run a training iteration with the desired batch size (for Habitat’s profiling pass).

One potential approach to making predictions for batch sizes larger than what can run on the origin GPU is as follows. First, use Habitat to make iteration execution time predictions for multiple (e.g., three) different batch sizes that *do* fit on the origin GPU. Then, build a linear regression model on these predicted values to *extrapolate* to larger batch sizes. This approach is based on our prior work, where we observed an often linear relationship between the iteration execution time and batch size [107]. We leave the handling of models where only one batch size fits on the origin GPU to future work.

6.2 Additional Deep Learning Frameworks

Recall that Habitat predicts the execution time of operations using either (i) wave scaling or (ii) pre-trained MLPs, depending on whether the operation is kernel-alike or kernel-varying. Therefore, as long as Habitat has information about a DNN’s operations and their parameters (e.g., batch size, number of channels), Habitat will be able to apply its techniques to make execution time predictions for a different GPU. Ultimately this means that adding support for other deep learning frameworks (e.g., TensorFlow or MXNet) boils down to extracting the underlying operations that run during a training iteration and sending the operations to Habitat (i.e., extracting the computation graph). Since the other major deep learning frameworks (TensorFlow and MXNet) both already use computation graphs internally [1, 19], we believe that adding support for them would be straightforward to implement.

6.3 Additional Hardware Accelerators

As described in Section 1, there are also other hardware options available beyond GPUs that can be used for training (e.g., the TPU [45], AWS Trainium [10], and Gaudi [36]). Therefore, a natural opportunity for future work is to explore execution time predictions for these other hardware accelerators. We outline two challenges that arise when going beyond GPUs, as well as two examples of ways that Habitat’s guiding principles can be applied to these prediction tasks.

Challenges. First, specialized deep learning accelerators may have a different hardware architecture when compared to GPUs—necessitating different performance modeling techniques. For example, the TPU uses a systolic array [14, 45] whereas GPUs are general-purpose SMT processors [85]. Second, accelerators such as the TPU rely on *tensor compilers* (e.g., XLA [34] or JAX [13]) to produce executable code from the high-level DNN model code written by an end-user. The compiler may apply optimizations that change the oper-

ations. These changes make the high-level operation-based analysis that Habitat performs more difficult to realize.

Opportunities. Despite these challenges, we believe that there are also opportunities to apply Habitat’s key idea of leveraging runtime-based information from one accelerator to predict the execution time on a different accelerator. For example, as of June 2021, Google makes two versions of the TPU available for rent (v2 and v3) [33] and has announced the v4 [51]. Execution times measured on the TPU v2 could potentially be used to make execution time predictions on the v3 and v4 and vice-versa. Similarly, assuming that the AWS Trainium also uses a systolic array,¹⁰ it may also be possible to leverage execution time measurements on the TPU to make execution time predictions for the Trainium and vice-versa.

7 Related Work

The key difference between Habitat and existing DNN performance modeling techniques for GPUs [46, 87, 88] is in how Habitat makes execution time predictions. Habitat takes a hybrid *runtime-based approach*; it uses information recorded at runtime on one GPU along with hardware characteristics to *scale* the measured kernel execution times onto different GPUs through either (i) wave scaling, or (ii) pre-trained MLPs. In contrast, existing techniques use analytical models [87, 88] or rely *entirely* on machine learning techniques [46]. The key advantage of Habitat’s hybrid scaling approach is that wave scaling works “out of the box” for all kernel-alike operations (i.e., operations implemented using the same kernels on different GPUs). Ultimately, this advantage means that new analytical or machine learning models do not have to be developed each time a new kernel-alike operation is introduced.

DNN performance models for different hardware. There exists prior work on performance models for DNN training on GPUs [46, 87, 88], CPUs [100], and TPUs [47]. As described above, Habitat is fundamentally different from these works because it takes a hybrid *runtime-based approach* when making execution time predictions. For example, Paleo [88] (i) makes DNN operation execution time predictions using *analytical models* based on the number of floating point operations (FLOPs) in a DNN operation, and (ii) uses heuristics to select the kernels used to implement kernel-varying operations (e.g., convolution). However, an operation’s execution time is not solely determined by its number of FLOPs, and using heuristics to select an analytical model cannot always capture kernel-varying operations correctly. This is because proprietary closed-source kernel libraries (e.g., cuDNN [21, 74], cuBLAS [77]) may select different kernel(s) to use by running benchmarks on the target GPU [44, 75].

Performance models for compilers. A complementary body of work on performance modeling is motivated by the

¹⁰The AWS Inferentia [9, 108], a related accelerator, uses a systolic array architecture [92]. So we believe that this is a reasonable assumption to make.

needs of compilers: predicting how *different implementations* of some high-level functionality perform on the *same hardware*. These models were developed to aid in compiling high-performance (i) graphics pipelines [2], (ii) CPU code [54], and (iii) tensor operators for deep learning accelerators [20, 47]. These models have fundamentally different goals compared to Habitat, which is a technique that predicts the performance of *different GPUs* running the *same high-level code*.

General scalability predictions. Wave scaling is similar in spirit to ESTIMA [17], since both use the idea of making measurements on one system to make performance predictions for a different system. However, ESTIMA is a scalability predictor for CPU programs. Wave scaling instead targets GPU kernels, which run under a different execution model when compared to CPU programs.

Repetitiveness of DNN training. Prior work leverages the repetitiveness of DNN training computation to optimize distributed training [43, 48, 60], schedule jobs in a cluster [18, 61, 103], and to apply DNN compiler optimizations [94]. The key difference between these works and Habitat is that they apply optimizations on the *same* hardware configuration. Habitat exploits the repetitiveness of DNN training to make performance predictions on *different* hardware configurations.

DNN benchmarking. A body of prior work focuses on benchmarking DNN training [3, 23, 53, 109]. While these works provide DNN training performance insights, they do so only for a *fixed* set of DNNs and hardware configurations. In contrast, Habitat analyzes DNNs in *general* and provides performance *predictions* on different GPUs to help users make informed GPU selections.

8 Conclusion

We present *Habitat*: a new runtime-based library that uses wave scaling and MLPs as execution time predictors to help deep learning researchers and practitioners make *informed cost-efficient* decisions when selecting a GPU for DNN training. The key idea behind Habitat is to leverage information collected at runtime on one GPU to help predict the execution time of a DNN training iteration on a different GPU. We evaluate Habitat and find that it makes cross-GPU iteration execution time predictions with an overall average error of 11.8% on ResNet-50, Inception v3, the Transformer, GNMT, and DCGAN. Finally, we present two case studies where Habitat correctly predicts that (i) optimizing for cost-efficiency would lead to selecting a different GPU for GNMT, and (ii) that the V100 does not offer significant performance benefits over a common desktop-class GPU (the 2080Ti) for DCGAN. We have also open sourced Habitat (github.com/geoffxy/habitat) to benefit both the deep learning and systems communities [105, 106].

Acknowledgments

We thank our shepherd, Marco Canini, and the anonymous reviewers for their feedback. We also thank (in alphabetical order) Moshe Gabel, James Gleeson, Anand Jayarajan, Xiaodan Tan, Alexandra Tsvetkova, Shang Wang, Qionsi Wu, and Hongyu Zhu. We thank all members of the [EcoSystem research group](#) for the stimulating research environment they provide. This work was supported by a QEII-GSST, Vector Scholarship in Artificial Intelligence, Snap Research Scholarship, and an NSERC Canada Graduate Scholarship – Master’s (CGS M). This work was also supported in part by the NSERC Discovery grant, the Canada Foundation for Innovation JELF grant, the Connaught Fund, an Amazon Research Award, and a Facebook Faculty Award. Computing resources used in this work were provided, in part, by the Province of Ontario, the Government of Canada through CIFAR, and companies sponsoring the Vector Institute www.vectorinstitute.ai/partners.

References

- [1] 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 Symposium on Operating Systems Design and Implementation (OSDI’16)*, 2016.
- [2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Transactions on Graphics (TOG)*, 38(4), 2019.
- [3] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Fathom: Reference Workloads for Modern Deep Learning Methods. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC’16)*, 2016.
- [4] Advanced Micro Devices, Inc. HBM2 - High Bandwidth Memory-2, 2015. <https://www.amd.com/system/files/documents/high-bandwidth-memory-hbm.pdf>.
- [5] Advanced Micro Devices, Inc. AMD Ryzen Threadripper 1950X Processor, 2017. <https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-1950x>.
- [6] Advanced Micro Devices, Inc. AMD EPYC™ 7371 Processor, 2020. <https://www.amd.com/en/products/cpu/amd-epyc-7371>.
- [7] Amazon, Inc. Amazon EC2 Instance Types, 2020. <https://aws.amazon.com/ec2/instance-types/>.
- [8] Amazon, Inc. Amazon SageMaker, 2021. <https://aws.amazon.com/sagemaker/>.
- [9] Amazon, Inc. AWS Inferentia, 2021. <https://aws.amazon.com/machine-learning/inferentia/>.
- [10] Amazon, Inc. AWS Trainium, 2021. <https://aws.amazon.com/machine-learning/trainium/>.
- [11] Ondřej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Varvara Logacheva, Christof Monz, Matteo Negri, Aurelie Neveol, Mariana Neves, Martin Popel, Matt Post, Raphael Rubino, Carolina Scarton, Lucia Specia, Marco Turchi, Karin Verspoor, and Marcos Zampieri. Findings of the 2016 Conference on Machine Translation. In *Proceedings of the First Conference on Machine Translation (WMT’16)*, 2016.
- [12] Léon Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. In *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT’10)*, 2010.
- [13] 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. <http://github.com/google/jax>.
- [14] Richard P Brent and Hsiang-Tsung Kung. Systolic VLSI Arrays for Polynomial GCD Computation. *IEEE Transactions on Computers*, 100(8):731–736, 1984.
- [15] Canonical Ltd. Ubuntu 18.04 LTS (Bionic Beaver), 2018. <http://releases.ubuntu.com/18.04/>.
- [16] Cerebras. Cerebras, 2020. <https://www.cerebras.net>.
- [17] Georgios Chatzopoulos, Aleksandar Dragojević, and Rachid Guerraoui. ESTIMA: Extrapolating Scalability of in-Memory Applications. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP’16)*, 2016.
- [18] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing Efficiency and Fairness in Heterogeneous

- GPU Clusters for Deep Learning. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys'20)*, 2020.
- [19] 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. In *Proceedings of the 2016 NeurIPS Workshop on Machine Learning Systems*, 2016.
- [20] 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.
- [21] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *arXiv*, abs/1410.0759, 2014.
- [22] Soumith Chintala. Deep Convolution Generative Adversarial Networks, 2020. <https://github.com/pytorch/examples/tree/master/dcgan/>.
- [23] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. DAWN Bench: An End-to-End Deep Learning Benchmark and Competition. In *Proceedings of the NeurIPS Workshop on Machine Learning Systems*, 2017.
- [24] PyTorch Contributors. torchvision, 2021. <https://github.com/pytorch/vision>.
- [25] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc' Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems 25 (NeurIPS'12)*, 2012.
- [26] 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 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL'19)*, 2019.
- [27] Docker, Inc. Docker, 2020. <https://www.docker.com/>.
- [28] Peter N. Glaskowsky. NVIDIA's Fermi: The First Complete GPU Computing Architecture. Whitepaper, NVIDIA, 2009.
- [29] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [30] Google, Inc. Google Cloud N1 Machine Types, 2020. https://cloud.google.com/compute/docs/machine-types#n1_machine_types.
- [31] Google, Inc. GPUs on Compute Engine, 2020. <https://cloud.google.com/compute/docs/gpus>.
- [32] Google, Inc. Google Cloud Vertex AI, 2021. <https://cloud.google.com/vertex-ai/>.
- [33] Google, Inc. Supported TPU Versions, 2021. <https://cloud.google.com/tpu/docs/supported-tpu-versions>.
- [34] Google, Inc. XLA: Optimizing Compiler for Machine Learning, 2021. <https://www.tensorflow.org/xla>.
- [35] Graphcore. Graphcore, 2020. <https://www.graphcore.ai>.
- [36] Habana Labs. Habana Labs, 2020. <https://habana.ai>.
- [37] 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'16)*, 2016.
- [38] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.
- [39] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. Densely Connected Convolutional Networks. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*, 2017.
- [40] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems 32 (NeurIPS'19)*, 2019.
- [41] Intel Corporation. Intel Xeon Processor E5-2680, 2020. <https://ark.intel.com/content/www/us/en/ark/products/91754/intel-xeon-processor-e5-2680-v4-35m-cache-2-40-ghz.html>.
- [42] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of*

the 32nd International Conference on International Conference on Machine Learning (ICML'15), 2015.

- [43] 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 (MLSys'19)*, 2019.
- [44] Marc Jorda, Pedro Valero-Lara, and Antonio J Peña. Performance Evaluation of cuDNN Convolution Algorithms on NVIDIA Volta GPUs. *IEEE Access*, 7:70461–70473, 2019.
- [45] 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 Gotipati, 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-Datcenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*, 2017.
- [46] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Stephen McGough. Predicting the Computational Cost of Deep Learning Models. In *Proceedings of the 2018 IEEE Conference on Big Data (BigData'18)*, 2018.
- [47] Samuel J. Kaufman, Phitchaya Mangpo Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. A Learned Performance Model for Tensor Processing Units. In *Proceedings of the 4th Conference on Machine Learning and Systems (MLSys'21)*, 2021.
- [48] 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 14th EuroSys Conference (EuroSys'19)*, 2019.
- [49] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference for Learning Representations (ICLR'15)*, 2015.
- [50] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25 (NeurIPS'12)*, 2012.
- [51] Naveen Kumar. Google breaks AI performance records in MLPerf with world's fastest training supercomputer, 2020. <https://cloud.google.com/blog/products/ai-machine-learning/google-breaks-ai-performance-records-in-mlperf-with-worlds-fastest-training-supercomputer>.
- [52] Lambda Labs Inc. Lambda: Deep Learning Workstations, Servers, Laptops, 2020. <https://lambdalabs.com>.
- [53] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. MLPerf Training Benchmark. In *Proceedings of the 3rd Conference on Machine Learning and Systems (MLSys'20)*, 2020.
- [54] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (ICML'19)*, 2019.
- [55] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed Precision Training. In *Proceedings of the 6th International Conference on Learning Representations (ICLR'18)*, 2018.
- [56] Micron Technology, Inc. GDDR5, 2015. <https://www.micron.com/products/graphics-memory/gddr5>.
- [57] Micron Technology, Inc. GDDR6, 2017. <https://www.micron.com/products/graphics-memory/gddr6>.

- [58] Microsoft Corporation. Azure Machine Learning, 2021. <https://azure.microsoft.com/services/machine-learning/>.
- [59] Ioannis Mitliagkas, Ce Zhang, Stefan Hadjis, and Christopher Ré. Asynchrony Begets Momentum, with an Application to Deep Learning. In *54th Annual Allerton Conference on Communication, Control, and Computing (Allerton'16)*, 2016.
- [60] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th Symposium on Operating Systems Principles (SOSP'19)*, 2019.
- [61] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, 2020.
- [62] NVIDIA Corporation. NVIDIA Pascal P100, 2016. <https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf>.
- [63] NVIDIA Corporation. NVIDIA Tesla P100. Whitepaper, NVIDIA, 2016.
- [64] NVIDIA Corporation. NVIDIA GeForce GTX 1080Ti, 2017. <https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080-ti>.
- [65] NVIDIA Corporation. NVIDIA Quadro P4000, 2017. <https://www.pny.com/nvidia-quadro-p4000>.
- [66] NVIDIA Corporation. NVIDIA Tesla V100, 2017. <https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf>.
- [67] NVIDIA Corporation. NVIDIA Tesla V100. Whitepaper, NVIDIA, 2017.
- [68] NVIDIA Corporation. NVIDIA TITAN Xp, 2017. <https://www.nvidia.com/en-us/titan/titan-xp>.
- [69] NVIDIA Corporation. NVIDIA GeForce RTX 2070, 2018. <https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2070>.
- [70] NVIDIA Corporation. NVIDIA GeForce RTX 2080Ti, 2018. <https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti/>.
- [71] NVIDIA Corporation. NVIDIA Tesla T4, 2018. <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf>.
- [72] NVIDIA Corporation. NVIDIA Turing Architecture. Whitepaper, NVIDIA, 2018. <https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf>.
- [73] NVIDIA Corporation. CUDA Runtime API – Event Management, 2019. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html.
- [74] NVIDIA Corporation. cuDNN Developer Guide, 2019. <https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html>.
- [75] NVIDIA Corporation. cuDNN Developer Guide: cudnnFindConvolutionForwardAlgorithm, 2019. <https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html#cudnnFindConvolutionForwardAlgorithm>.
- [76] NVIDIA Corporation. CUPTI Documentation, 2019. <https://docs.nvidia.com/cupti/Cupti/index.html>.
- [77] NVIDIA Corporation. cuBLAS: Dense Linear Algebra on GPUs, 2020. <https://developer.nvidia.com/cublas>.
- [78] NVIDIA Corporation. NVIDIA A100, 2020. <https://www.nvidia.com/en-us/data-center/a100>.
- [79] NVIDIA Corporation. NVIDIA Ampere Architecture In-Depth, 2020. <https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/>.
- [80] NVIDIA Corporation. NVIDIA CUDA Toolkit, 2020. <https://developer.nvidia.com/cuda-toolkit>.
- [81] NVIDIA Corporation. NVIDIA Data Center Deep Learning Product Performance, 2020. <https://developer.nvidia.com/deep-learning-performance-training-inference>.
- [82] NVIDIA Corporation. NVIDIA GeForce RTX 3090, 2020. <https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3090/>.
- [83] NVIDIA Corporation. NVIDIA GPU Cloud Virtual Machine Image Release Notes, 2020. <https://docs.nvidia.com/ngc/ngc-ami-release-notes/>.

- [84] NVIDIA Corporation. Quadro RTX 6000 Graphics Card, 2020. <https://www.nvidia.com/en-us/design-visualization/quadro/rtx-6000/>.
- [85] NVIDIA Corporation. CUDA Programming Guide, 2021. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
- [86] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32 (NeurIPS'19)*, 2019.
- [87] Ziqian Pei, Chensheng Li, Xiaowei Qin, Xiaohui Chen, and Guo Wei. Iteration Time Prediction for CNN in Multi-GPU Platform: Modeling and Analysis. *IEEE Access*, 7:64788–64797, 2019.
- [88] Hang Qi, Evan R. Sparks, and Ameet Talwalkar. Paleo: A Performance Model for Deep Neural Networks. In *Proceedings of the 5th International Conference on Learning Representations (ICLR'17)*, 2017.
- [89] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. In *Proceedings of the 4th International Conference on Learning Representations (ICLR'16)*, 2016.
- [90] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*. MIT Press, 1986.
- [91] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.
- [92] Julien Simon. Announcing availability of Inf1 instances in Amazon SageMaker for high performance and cost-effective machine learning inference, 2020. <https://aws.amazon.com/blogs/machine-learning/announcing-availability-of-inf1-instances-in-amazon-sagemaker-for-high-performance-and-cost-effective-machine-learning-inference/>.
- [93] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR'15)*, 2015.
- [94] Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, and Lidong Zhou. Astra: Exploiting Predictability to Optimize Deep Learning. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, 2019.
- [95] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and Policy Considerations for Deep Learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL'19)*, 2019.
- [96] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to Sequence Learning with Neural Networks. In *Proceedings of Advances in Neural Information Processing Systems 27 (NeurIPS'14)*, 2014.
- [97] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. *arXiv*, abs/1512.00567, 2015.
- [98] TechPowerUp. TechPowerUp GPU Database (P4000 and 2070), 2020. <https://www.techpowerup.com/gpu-specs/>.
- [99] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All You Need. In *Advances in Neural Information Processing Systems 30 (NeurIPS'17)*, 2017.
- [100] Andre Viebke, Sabri Pllana, Suejb Memeti, and Joanna Kolodziej. Performance Modelling of Deep Learning on Intel Many Integrated Core Architectures. *arXiv*, abs/1906.01992, 2019.
- [101] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures. *Communications of the ACM (CACM)*, 52(4):65–76, 2009.
- [102] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado,

Macduff Hughes, and Jeffrey Dean. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv*, abs/1609.08144, 2016.

- [103] 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 *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI'18)*, 2018.
- [104] Fisher Yu, Yinda Zhang, Shuran Song, Ari Seff, and Jianxiong Xiao. LSUN: Construction of a Large-scale Image Dataset using Deep Learning with Humans in the Loop. *arXiv*, abs/1506.03365, 2015.
- [105] Geoffrey X. Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training (Code), 2021. <https://doi.org/10.5281/zenodo.4885489>.
- [106] Geoffrey X. Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. Habitat Pre-Trained Models and Kernel Metadata, 2021. <https://doi.org/10.5281/zenodo.4876277>.
- [107] Geoffrey X. Yu, Tovi Grossman, and Gennady Pekhimenko. Skyline: Interactive In-Editor Computational Performance Profiling for Deep Neural Network Training. In *Proceedings of the 33rd ACM Symposium on User Interface Software and Technology (UIST'20)*, 2020.
- [108] Hongbin Zheng, Sejong Oh, Huiqing Wang, Preston Briggs, Jiading Gai, Animesh Jain, Yizhi Liu, Rich Heaton, Randy Huang, and Yida Wang. Optimizing Memory-Access Patterns for Deep Learning Accelerators. In *Proceedings of the 2nd Compilers for Machine Learning Workshop at CGO 2020 (C4ML'20)*, 2020.
- [109] Hongyu Zhu, Mohamed Akrouf, Bojian Zheng, Andrew Pelegris, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. Benchmarking and Analyzing Deep Neural Network Training. In *Proceedings of the 2018 IEEE International Symposium on Workload Characterization (IISWC'18)*, 2018.
- [110] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. Daydream: Accurately Estimating the Efficacy of Optimizations for DNN Training. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC'20)*, 2020.

Zico: Efficient GPU Memory Sharing for Concurrent DNN Training

Gangmuk Lim
UNIST

Jeongseob Ahn
Ajou University

Wencong Xiao
Alibaba Group

Youngjin Kwon
KAIST

Myeongjae Jeon
UNIST

Abstract

GPUs are the workhorse in modern server infrastructure fueling advances in a number of compute-intensive workloads such as deep neural network (DNN) training. Several recent works propose solutions on sharing GPU resources across multiple concurrent DNN training jobs, but none of them address rapidly increasing memory footprint introduced by such job co-locations, which greatly limit the effectiveness of sharing GPU resources. In this paper, we present Zico, the first DNN system that aims at reducing the system-wide memory consumption for concurrent training. Zico keeps track of the memory usage pattern of individual training job by monitoring its progress on GPU computations and makes memory reclaimed from the job globally sharable. Based on this memory management scheme, Zico automatically decides a strategy to share memory among concurrent jobs with minimum delay on training while not exceeding a given memory budget such as GPU memory capacity. Our evaluation shows that Zico outperforms existing GPU sharing approaches and delivers benefits over a variety of job co-location scenarios.

1 Introduction

Recent advances in deep neural networks (DNNs) have made tremendous progress on a wide range of applications, including object detection [24], language model [11, 47], translation [40], and speech recognition [27]. As a number of new DNN models are being explored, developers take advantage of hardware accelerators to train the models, such as TPU [22] and GPU, which is the most popular choice. GPUs are the workhorse in server infrastructure and yet becoming highly contended resources at the same time [20, 43].

To utilize expensive GPU resources, efficient GPU sharing mechanisms have become indispensable. Prior work focuses on either *temporally* multiplexing GPU in its entirety [26, 43, 44] or *spatially* sharing compute units [10]. The temporal sharing is a software mechanism that dedicates both compute cores and memory in GPU solely to single training

for a time quantum (e.g., 1 minute). Despite the good flexibility, this approach often cannot efficiently utilize GPU resources. For example, most compute resources are left idle for common translation models such as GNMT [42] and language models such as RHN [47]. These training algorithms include a number of RNN modules [28], such as LSTM [12] and GRU [8] networks, which exhibit a small degree of data parallelism to GPU, causing under-utilization of GPU resources. As a different approach, the spatial sharing can provide better throughput than the temporal sharing as long as a single training job does not fully saturate GPU compute resources [44]. However, a limitation in applying the spatial sharing is the working set size of concurrent jobs, which grows substantially with the job co-location. If the working set does not fit in GPU memory, the system has to kill a job or swap GPU memory to the host, which overshadows the performance benefit of the spatial sharing. Therefore, to make the spatial sharing widely applicable, it is essential to reduce the memory footprint of co-located training jobs.

We observe that sharing intermediate data generated during co-located training jobs significantly reduces the total memory footprint. Training is a highly iterative procedure first navigating layers in order (forward pass) and then the same layers in *reverse* order (backward pass) for each batch of input data. During the training procedure, intermediate outputs from model layers called *feature maps* dominate memory footprint [18, 32]. Feature maps are generated in each layer during the forward pass and later consumed in the backward pass to update the layer. Due to the regular bi-directional execution, memory consumption in a single training job commonly exhibits a *cyclic pattern* — memory consumption gradually increases in the forward pass and then decreases in the backward pass. Thus, a simple yet effective strategy to save memory consumption is creating a large GPU memory pool and elastically sharing the memory pool for concurrent training jobs. To increase sharing opportunity, coordination of concurrent training jobs is needed to make them run different passes, e.g., forward pass for job A (increasing its working set) and backward pass for job B (decreasing its working set). This

approach results in memory allocations of a job to happen simultaneously with memory deallocations of the other job, efficiently reducing the *system-wide* memory footprint.

Despite that the sharing idea is plausible, the way today's DNN frameworks execute training on GPU poses significant challenges. Current frameworks are mainly designed for a *solo training* case. Following dataflow dependency, they allocate the memory required for each DNN kernel computation ahead of time and issue as many kernels as possible to the GPU stream, i.e., work queue per job for its GPU computations, in order to saturate the GPU's internal compute resources. This leads to GPU computations *asynchronous* and in parallel with CPU processing. Thus, the platforms are *unaware* of progress on the GPU computations and when memory is indeed consumed by GPU. Without proper handling of the asynchrony, shared memory does not guarantee correctness such as preventing memory corruptions in shared untapped memory of a waiting kernel by other training jobs.

In this paper, we propose Zico, a DNN platform that enables efficient memory sharing for concurrent training. Zico retrofits a widely used DNN platform, TensorFlow, to maximize the overall throughput of concurrent training. The goal of Zico is finding the best coordinated executions of concurrent training to fully utilize GPU computational and memory resources. To achieve the goal, (i) Zico accurately monitors computational progress of training jobs. Based on that, Zico allocates and deallocates memory for DNN kernels, informing memory usage patterns closer to GPU's view. (ii) Zico incorporates runtime information (e.g., iteration times, memory usage patterns, and GPU memory limit) and executes a job scheduler, called *scrooge scheduler*, to efficiently steer concurrent jobs to utilize the shared memory pool. (iii) Zico efficiently organizes the entire GPU space as an elastic shared memory pool to support scrooge scheduler.

To detect computational progress of asynchronous kernels, Zico leverages a specific kernel called CUDA event, which notifies progress of GPU kernels. Zico uses CUDA event to identify allocation and release time of memory used by a GPU kernel. Based on the information, Zico executes our novel scrooge scheduler to forecast the memory consumption trend of concurrent training at the iteration boundary and introduces the minimum stall time on each iteration. Nevertheless, the memory usage trend of the co-scheduled jobs varies according to how they interfere each other in the use of GPU compute units. To apply their dynamic behaviors, scrooge scheduler refines decisions based on feedback collected at runtime.

Zico organizes the memory pool as a collection of chunks called *regions* and separates their uses based on data characteristics. DNN training generates several types of data as tensors, categorized mainly as *ephemeral* tensors with high occurrences and *long-lived* tensors like feature maps which constitute the most memory footprint. By separating regions by type, Zico ensures that memory stored with feature maps does not interfere with other transient data while making their

demands follow the cyclic pattern of training iteration. This design choice allows our scheduling decisions to be applied with little disruption without losing sharing opportunity.

Being prototyped on a popular DNN framework, TensorFlow, we evaluate Zico experimentally using six models ranging from translation to object detection on V100 and RTX 2080 Ti GPUs. The results show that Zico enables effective memory sharing over a wide range of memory consumption trends. For high memory footprint, Zico is up to 8.3x and 1.6x faster than traditional spatial sharing and temporal sharing approaches, respectively, especially when concurrently training non-identical models. Furthermore, for low memory footprint, where no stall on concurrent training is needed, Zico behaves similarly to traditional spatial sharing and is up to 1.6x faster than temporal sharing. Overall, Zico achieves speedups, regardless of whether concurrent training is based on the same or distinct models.

2 Background

2.1 Deep Neural Network Training

The training process typically relies on iterative optimization methods like stochastic gradient descent (SGD) [13], Momentum [39], and Adam [23]. In each *iteration*, *forward pass* (FP) is followed by *backward pass* (BP) on a *batch* of training dataset. During FP, by computing on the layer's input, weights, and bias, each layer outputs *feature maps* to be used as an input to the next downstream layer. At the end of FP, the last layer produces a loss representing the accuracy of the current model on the input batch. Using the loss value, BP computes the gradients by flowing the layers in reverse order and aggregates the gradient values to update model *parameters* (i.e., layers' *weights* and *bias*). On finishing BP, the training repeats FP and BP on the next batch. As the batch size is usually fixed, the computation load and the memory usage characteristic are usually very similar across iterations [15, 43].

It is widely known that model parameters occupy only a small fraction of memory, and the majority is consumed to store feature maps generated in the FP computation [18, 32, 43]. BP needs feature maps to calculate the gradients at each layer. Hence, unless recomputed [6, 14, 19], feature maps are usually kept on memory for a long time until they are no longer accessed in BP. The amount of memory consumption is determined by several factors, such as the number of layers, layer size and type, input batch size, etc. There is also other intermediate data training iteration creates, e.g., *gradient maps* that represent the output of each layer during BP, local data local in each kernel, etc. They are *all* ephemeral as memory is released soon after its allocation [18, 32]. For brevity, we assume all memory allocations in DNN training are based on *tensors*.

2.2 GPU Sharing Use Cases

Users run training either on shared GPU clusters or on dedicated servers. For both cases, GPU sharing is becoming a fundamental technique to better utilize GPU resources. In this subsection, we introduce two specific scenarios that can take advantage of sharing GPUs.

Hyperparameter tuning (inter-job). With the increasing popularity of applications fueled by DNN, a number of new models are being developed by DNN practitioners every day. A model for developing exposes many high-level properties, e.g., learning rate and momentum, as hyperparameters that need to be optimized. This task is known as *hyperparameter tuning* [3]. As hyperparameters constitute a large search space, there are several popular tools such as Hyperdrive [35] and HyperOpt [4] that automate hyperparameter optimization and construct a new model with the best (or desired) quality for users. These tools usually generate a large number of closely related training jobs (as much as 100s [26, 43]) that explore a different set of hyperparameters for the same reference model. Nevertheless, spatial GPU sharing has greater performance potential for this workload, as discussed in Section 7.

Hyperparameter tuning jobs dominate training workloads run atop shared GPU clusters [20, 26, 43]. To get them timely done on heavily contended GPUs, prior works propose several techniques such as temporal and spatiotemporal sharing to apportion a single GPU over multiple training jobs [43, 44].

Gradient accumulation (intra-job). Gradient accumulation is a promising method to speed up model convergence when hyperparameters other than batch size are stabilized. It runs a set of consecutive mini-batches and accumulates the gradients of those mini-batches before updating model parameters. The essential goal is to give an illusion of training on a large batch that better improves convergence without oversubscribing GPU memory by using small mini-batches. A common practice has been to process these mini-batches sequentially.

Nonetheless, efficient spatial GPU sharing can offer sharing incentive to concurrent training on mini-batches. One might wonder whether spatial sharing on this training is indeed plausible. Based on our observation, translation or speech recognition models (e.g., GNMT [42]) often underutilize GPU compute resources, making it beneficial to share GPU computes. On top of it, our system supports highly effective sharing of GPU memory, enabling in concurrent training each to use a mini-batch size slightly smaller, if not the same, than the original mini-batch size. Altogether, our system opens up a new opportunity to speed up training for gradient accumulation, which we will discuss in Section 7.

2.3 Spatial GPU Sharing

NVIDIA has developed Multi-Process Service (MPS) [10], an alternative way to share GPU among multiple CUDA processes. With MPS, NVIDIA V100 GPU supports up to 48 pro-

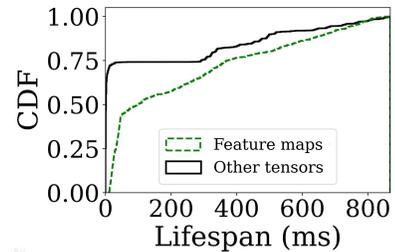


Figure 1: Cumulative distribution of NASNet tensor lifespan.

cesses to run concurrently on a single GPU, with each process assigned with separate GPU compute resources, i.e., SMs [10]. In NVIDIA A100 [29], a newer generation, GPU sharing architecture further partitions HW paths in the memory system, e.g., memory controllers and address buses, to prevent the concurrent processes from interfering with each other when demands for memory bandwidth are high. NVIDIA’s GPU sharing mechanisms have two commonalities. First, they are mainly designed for sharing “compute resources” spatially. Second, they attribute GPU sharing to demands for protection among untrusted users requiring strong isolation.

Since not all use cases require strong isolation among training jobs, e.g., hyperparameter tuning driven by a single user [26], recent work supports a spatial GPU sharing similar to MPS in a single process domain [44]. Regardless of protection level, the underlying mechanism enabling spatial sharing within GPU is very similar, if not the same — and so is the resulting performance.

3 Challenges for Memory Sharing

GPU has a limited amount of HW resources, requiring it to be used in high efficiency. As GPU’s compute and memory resources are shared to run concurrent training limiting per-training resource capacity, it is crucial to thoroughly understand the current practices in DNN frameworks and uncover challenges for spatial GPU training. In this section, we discuss three major challenges to address in Zico.

3.1 Memory Bloating

Major DNN frameworks [1, 5, 31] typically maintain feature maps in memory until they are no longer accessed. As discussed earlier, feature maps have a relatively longer lifespan between the first access and the last access, making the most of in-use memory consumed to store the feature maps. Figure 1 compares cumulative distributions of lifespans for feature maps and other data in NASNet training. As the figure shows, feature maps exhibit longer lifespans with 134ms on average and 234ms as median value as opposed to 18ms on average and 2ms as median value in the other data. We further investigate cumulative distributions of tensor sizes of the two data types, showing that feature maps are larger. In consequence, as shown in Figure 2, feature maps lead to the peak

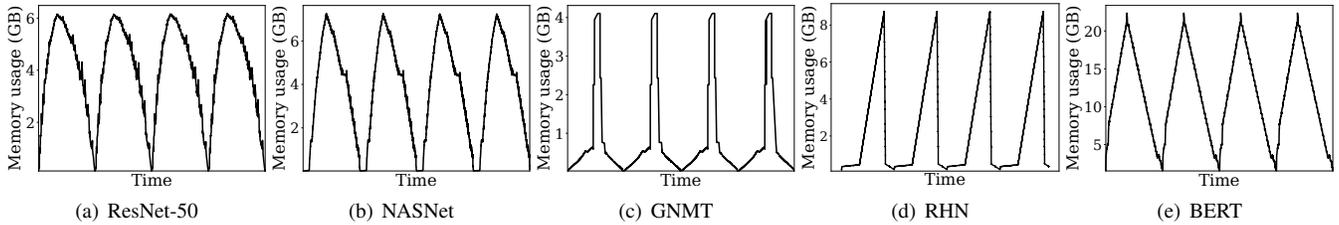


Figure 2: Memory usage patterns for different DNN models over time.

memory consumption substantially higher than the minimum that corresponds to the model size in each iteration. We call this issue *memory bloating*.

Traditional spatial GPU sharing mechanisms are vulnerable to memory bloating. DNN frameworks like TensorFlow do not tend to be designed for memory sharing with internal memory manager maintaining a *local pool* of memory for single training. Thus, no memory release to GPU happens. This results in peak memory usages from concurrent training, all adding up and consequently saturating GPU memory even in the modest memory footprint for individual training. To illustrate, let us consider concurrent training of two identical models, with each demanding more than half of GPU memory as an example. In this case, the memory demand across the two local memory pools exceeds GPU memory capacity, beginning to take advantage of CPU-side memory as a swap space. To facilitate this, recent NVIDIA GPUs, including V100, provide a feature known as Unified Virtual Memory (UVM), which is transparent to DNN platforms.

We found that using UVM for DNN training is currently costly and severely affects overall performance despite great flexibility. To confirm the effect, we compare throughput between solo training versus concurrent training for ResNet-50 using NVIDIA V100 when a training job occupies 70% GPU memory. To make the comparison fair, we configure a single training job to use 50% GPU resources set by MPS. There is a dramatic throughput degradation in concurrent training (i.e., 8 times slower) as it suffers from GPU memory oversubscription. Therefore, we should decrease the risk of GPU memory being used up during concurrent training.

3.2 Workload Variability

As an additional challenge, we explain a workload characteristic that makes memory optimization for spatial GPU sharing fundamentally complicated. Although all models follow a cyclic pattern in memory usage, as shown in Figure 2, memory usage patterns are inherently different across models. For example, ResNet-50 has a beefy shape in which memory bloating appears for a fairly large time duration. In contrast, GNMT has a lean shape in which peak memory appears for a short period and quickly disappears. Therefore, such variability must be taken into account in designing a scheduling policy for concurrent training.

Nonetheless, we take advantage of an observation that a

similar memory usage pattern repeats over iterations for a single model. DNN frameworks require that computation kernels be ordered in a specific way, e.g., following topological sort, thus keeping the corresponding memory operations ordered across iterations [32]. So, we believe this determinism is prevalent in most of the training tasks.

3.3 Asynchrony with GPU Processing

CPU processing in DNN frameworks is in parallel with GPU computations. Before issuing a kernel to a GPU stream, the memory manager in the platform allocates tensors required for the kernel computation. After issuing the kernel and returning immediately, CPU processing brings back its control and can do any subsequent task asynchronously with GPU computations. Meanwhile, GPU may or may not execute the issued kernel depending on whether earlier kernels are still pending or not.

Driven by this GPU-specific property, DNN frameworks produce a static schedule of DNN kernels mainly customized for single training, in which kernel operations are *ordered* based on dependencies every iteration. DNN frameworks usually allocate the memory required for kernel operations in sequence *ahead of time* while issuing *as many kernels as possible* to the stream to saturate GPU. This way of exercising GPU for single training has been common because there is no concern regarding correctness in memory allocations. Specifically, suppose we use a released tensor memory from a kernel to allocate it for the next kernel, even though the earlier kernel is not completed by GPU yet. In that case, memory corruption does not occur since kernels in the GPU stream are processed sequentially.

However, it is critical to make the CPU process keep track of GPU memory usage trends precisely to enable memory sharing for concurrent training. To illustrate the current limitation concretely, let us show effective memory observed in TensorFlow during training ResNet-50 in Figure 3. In the figure, training exhibits a cyclic pattern for a short period followed by a long pause due to kernels pending on GPU during an iteration. Unlike the CPU’s view, the actual memory usage trend from GPU’s view appears in Figure 2(a), which perfectly follows a continuous cycling pattern.

In summary, for concurrent training with multiple GPU streams, today’s approaches that make memory sharing decisions based on CPU’s view are vulnerable to memory cor-

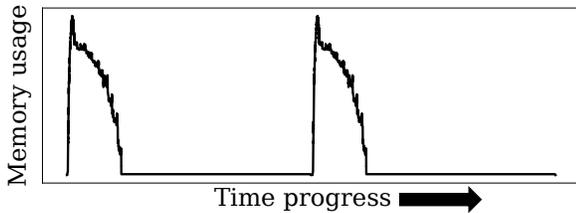


Figure 3: GPU memory usage from CPU view (ResNet-50).

ruption led by simultaneous accesses on the same memory address. This issue is the last challenge to address.

4 Design Overview

Zico aims at providing efficient spatial GPU sharing by enabling coordinated job scheduling and GPU memory management for concurrent training. As a system currently built on TensorFlow, the framework keeps tracking the lifespans of memory used in each training and produces a schedule of concurrent training executions to avoid GPU memory oversubscription. We seek to achieve the following goals in the design of Zico:

- **High performance.** A single iteration runs for a short time duration from tens to hundreds of milliseconds. Thus, a modest overhead with memory sharing in each iteration can manifest as a long delay in the entire training. We should attempt to minimize such overhead and achieve high performance.
- **Wide model support.** Section 3.2 demonstrates a wide range of patterns in memory demand across models during training. Our memory sharing strategy should be general, not restricted to specific memory patterns. The concurrent training may not even expose similar or the identical models.
- **Common approach.** Our approaches are mainly compatible with DNN platforms that express a job execution as a computation graph ahead of time, e.g., TensorFlow [1], Caffe [21], and MXNet [5]. The other platforms are based on imperative programming and train a model without constructing a computation graph, namely, the eager mode. The memory-aware scheduler in Zico makes decisions based only on observed memory demands at runtime, making it also applicable to the eager mode.

Zico has two key system modules as shown in Figure 4: (i) *scrooge scheduler*, a processing unit that orchestrates executions of concurrent training driven by memory demand patterns; and (ii) *memory manager*, a unified memory allocator that handles both inter- and intra-training memory requests.

Scrooge scheduler. Zico monitors GPU progress to capture precise memory usage in GPU for each training over time. On observing memory usage, Zico schedules concurrent jobs to achieve the best possible performance in training while putting the total memory consumption within a predefined budget, which does not exceed GPU memory capacity. This is a sophisticated problem to which a naive strategy rarely

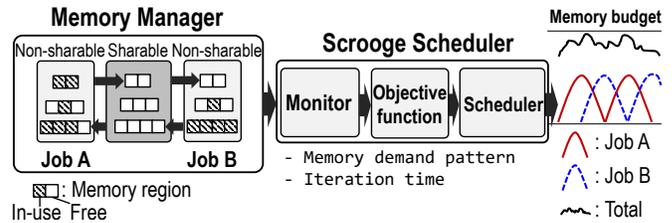


Figure 4: System architecture in Zico.

works. When memory is sufficient for running concurrent training without sharing, no coordination among jobs might be necessary. In contrast, when memory is tight, we need a schedule of concurrent training in order not to exceed the given memory budget. A simple approach towards saving memory consumption can be coordinating training jobs in such a way that forward pass execution (i.e., memory allocation phase) is overlapped with backward pass execution (i.e., memory deallocation phase) as much as possible. However, when memory demand patterns are beefy such as in Figure 2(a), the system-wide memory usage can blow up because the memory demand in the forward pass grows fast while the backward pass shrinks its memory demand relatively slowly.

To provide concrete guidance for memory sharing, we devise *scrooge scheduler* that facilitates an objective function helping forecast the system-wide memory demand in the future. Before starting a new iteration, the scheduler takes into account the lifespans of in-use memory regions, which are sharable memory units in Zico. Then, the scheduler predicts whether allowing the iteration immediately will consume memory less than the memory budget. If this is nearly impossible, the scheduler estimates the minimum stall time to be applied on the new iteration so that memory allocations in the forward pass to be issued later are safely fulfilled when more memory is available. It is essential to maintain precise region lifespan in order to make a correct decision in the scheduling. To meet this goal, Zico iteratively refines region lifespan based on feedback collected from prior iterations at runtime.

Scrooge scheduler is agnostic to programming model in DNN platforms and only relies on information on memory demand patterns. For this reason, Zico is able to perform spatial memory sharing as long as memory requests are appropriately clustered on regions and their lifespans can be estimated. As explained next, we facilitate tensor types to constitute regions for sharing, but many different ways are indeed possible — e.g., classification based on tensor access time intervals in eager mode [2].

Memory manager. Zico organizes the entire GPU memory space into a collection of *regions*, which is a contiguous memory space that stores a set of tensors in the same type. Using regions is a natural choice for us to mitigate the sharing overhead and to keep memory stored with feature maps not contended with other tensor data. Using regions further assists job scheduling decision to be made promptly. Scrooge scheduler makes use of memory demand pattern that changes dynami-

cally within an iteration. If we consider memory changes at the granularity of tensor, we may need to investigate too many time points along the way, putting a strain on the scheduler. Details will be discussed in Section 5.

The memory manager separates memory space into two areas, *sharable* and *non-sharable*. The sharable area represents currently unused memory that can be granted to any in-flight training in need of more space (mainly for feature maps), whereas the non-sharable area constitutes job-local memory pools. Zico analyzes the computational graph and extracts type information on tensors at runtime. During training iterations, allocation requests for feature map tensors are always served from their own regions first in the local memory pool. If regions in the local memory pool are used up, the memory manager assigns a new region from the sharable area. In general, feature maps demand the most regions in the system and these regions are mainly shared across concurrent training jobs.

The current design of Zico limits two training jobs to share a common memory area since many models we observe, including models in Figure 2, exhibit rather beefy memory demand patterns or high GPU utilization. For co-locating more than two jobs, a feasible approach is to organize the jobs into a group of pairs and schedule each pair independently with its own memory budget. This is a natural extension to Zico, so we leave it as a future work.

Protection level. We provide Zico as a single framework instance mainly for performance reason. Existing multi-process solutions such as MPS do not promise good performance for elastic memory space sharing across different processes. For example, to grant memory across two MPS processes, we require invoking CUDA APIs such as `cudaMalloc` and `cudaFree` quite frequently at each training iteration. Among these API, `cudaFree` is known to stall GPU’s pipelined execution upon invoked [9], making itself harmful if recklessly used. Our measurement also reveals that a single ResNet-50 training that allocates memory using these APIs becomes 7x slower than the training that allocates memory locally.

Note that key scenarios discussed in Section 2.2 are enough to train models in the same protection domain. Apart from it, we design Zico to be useful for a variety of scenarios as long as isolation is not a primary concern, e.g., the same tenancy with trusting users in a shared GPU cluster.

5 Scheduling Algorithm

In this section, we formalize the scheduling problem for concurrent training and introduce the *scrooge scheduler*. All related implementation details on how to obtain memory usage patterns are explained in Section 6.

5.1 Problem Definition

We make use of memory consumption at the region level to shape the memory pattern of a job. Despite the coarser-grained leveling of memory utilized by tensors, using regions still provides meaningful information to compute memory sharing potential. Since regions allocated for feature maps are the main target for sharing, in the problem formulation, we assume all regions for a job have the lifetime that spans the forward-backward passes for a single iteration.

Formally, we denote M regions required for an iteration of a training job as $\{R_i : 1 \leq i \leq M\}$ following the *allocation order*, with region R_i having two parameters to indicate its *lifespan*: $Time(R_i(a))$ for the allocation time and $Time(R_i(d))$ for the deallocation time. Assume at time moment T that there are K active regions in the system that indicate the sum of memory footprint of the concurrent jobs. To achieve efficient GPU memory provisioning, we need to *minimize the time delay* to be applied on each training iteration without overcommitting the system-wide memory budget C . This objective turns into the following formulation:

$$\underset{TimeShift}{\operatorname{argmin}} (Time(R_1(d)) - Time(R_1(a)) + TimeShift) \quad (1)$$

subject to

$$\sum_{i=1}^K Size(\hat{R}_i) \leq C \quad (2)$$

at any time T over a training iteration, where $\{Size(\hat{R}_i) : 1 \leq i \leq K\}$ are the sizes of active regions in the system.

Intuitively, Equation 1 represents that an iteration, whose duration corresponds to $Time(R_1(d)) - Time(R_1(a))$, must be delayed by the minimum *Timeshift*. Note that there are other costs such as model synchronization in distributed training that affect training time. They are mostly static [15, 25, 37, 45] and can be easily factored in.

5.2 Time Shift Model

From the perspective of memory sharing, the worst case possible is having forward passes of all training jobs run simultaneously. This may lead to no memory sharing opportunity as regions being freed out in the backward pass of a job may not be used by the other training job. Therefore, when it comes to exceeding the memory budget C , scrooge scheduler adds a time delay driven by the *TimeShift* parameter to a training iteration appropriately to ensure that memory allocations during the forward pass occur when enough memory is available.

Since DNN training is highly periodic and deterministic, when training on an apportioned GPU compute capacity is stabilized, we see almost no variations on the region lifespans over iterations. Moreover, for each iteration, allocation times for adjacent regions exhibit strong temporal dependency. In other words, the time interval between $Time(R_{i(a)})$ and

$Time(R_{i+1}(a))$ for a job is almost static and does not change drastically over iterations. Similarly, deallocations of any two consecutive regions have such strong temporal dependency. This observation suggests that *TimeShift* is the most effective when applied to the entire iteration, not an individual region. Suppose we postpone the allocation of an arbitrary region R_i by *TimeShift*. In that case, deallocations of the regions between R_1 and R_i will be all postponed by *TimeShift*, because allocations for the regions subsequent to R_i get delayed by *TimeShift* and temporal dependencies during deallocations are still preserved. This results in increasing overall memory usage for the iteration unnecessarily.

5.3 Memory Sharing Algorithm

Now, we explain how scrooge scheduler works to enable spatial memory sharing. Scrooge scheduler optimizes for the minimum possible iteration time based on Equation 1 at run-time. To solve the problem, the scheduling algorithm must address two challenges: C-1) The lifespan of the region, $L(R_i)$, changes according to how two training jobs execute under co-location; C-2) While $L(R_i)$ is changing, the schedule has to find an optimal *TimeShift* in Equation 1. Scrooge scheduler performs iterative searching steps to reach the goal. For C-1, scrooge scheduler introduces a feedback-driven online process in which the scheduler monitors lifespans of all R_i during the current step and updates them to use in the next step. For C-2, at each step, scrooge scheduler decides *TimeShift* of the co-located training jobs toward minimizing their iteration times. After several steps, the lifespans are adjusted enough and stabilized. *TimeShift* should be estimated in an iteration basis to determine when the current iteration has to start.

Profiling phase (The first search step). When a new training job is issued, scrooge scheduler initiates *profiling phase* during which it collects basic information on the new job. In particular, the scheduler runs the first iteration of the new job in an isolated manner. During this profile phase, scrooge scheduler identifies regions by type and obtains lifespan for each feature map region in the solo run¹.

Informed phase. After the profile phase, scrooge scheduler knows useful information for co-locating jobs. In this informed phase, for a new iteration to be started for a job (e.g., job A) with $TimeShift = 0$, the scheduler navigates through time progress using lifespan information and predicts if the memory constraint in Equation 2 is violated or not. If violated, the scheduler waits for time T , which leads to $TimeShift += T$, and does the prediction again. This time-shifting continues until the scheduler meets the memory constraint — this is guaranteed as the other co-located job (e.g., job B) will ultimately release all regions at the end of its backward pass.

To illustrate, for job A's forward pass, $Time(R_2^{Job A}(a)) - Time(R_1^{Job A}(a))$ indicates the time duration in which the

job A uses $Size(R_1^{Job A}(a))$ amount of memory, which corresponds to the allocation of the first region $R_1^{Job A}$. Likewise, $Time(R_N^{Job A}(a)) - Time(R_1^{Job A}(a))$ indicates the time taken for the entire forward pass in which job A gradually demands more memory by allocating regions from $R_1^{Job A}$ to $R_N^{Job A}$. In this way, scrooge scheduler can forecast the memory demand trend for job A's forward pass and similarly the trend for its backward pass.

To fulfill the condition in Equation 2, scrooge scheduler also needs to know the memory demand trend of the co-located job B. Assume that job B deallocates its R_{i+1} , i.e., $R_{i+1}^{Job B}$, when the scheduler attempts to schedule job A's forward pass. Then, scrooge scheduler knows that during $Time(R_{i-1}^{Job B}(d)) - Time(R_i^{Job B}(d))$, job B will use $\sum_i^i Size(R_i^{Job B})$ amount of memory. As such, as job B gradually deallocates its in-use regions from $R_i^{Job B}$ to $R_1^{Job B}$ in order over time, job B will ultimately release $\sum_i^i Size(R_i^{Job B})$ amount of memory after $Time(R_1^{Job B}(d)) - Time(R_i^{Job B}(d))$.

With this memory trend information on job A and job B, scrooge scheduler can decide during Job B's backward pass, if Job A can start by computing 1) the amount of memory required by Job A as time progresses and 2) the amount of memory released by job B as time progresses in terms of regions. This brings out the system-wide active regions as time progresses, which scrooge scheduler uses to predict if memory consumption will always fit in the defined memory budget, i.e., if Equation 2 is satisfied.

In the informed phase, we first use a memory budget which is lower than the actual memory budget C to calculate region lifespans under a conservative schedule that incurs a smaller interference among jobs and thus a less aggressive memory sharing. At this time, the execution of concurrent training is far from optimal, i.e., having long time shifts. From this point, scrooge scheduler iteratively optimizes the objective function. The scheduler gradually increases the lowered memory budget to allow more interference over time and keeps updating region lifespans until it reaches back the memory budget C . It is necessary to calculate region lifespans under co-location with such adaptation because co-scheduling jobs that together have >100% compute demands than the GPU's capacity will inadvertently interfere with the lifespan calculations.

Note that scrooge scheduler works regardless of training two heterogeneous models or when forward passes of the two jobs overlap. Although scheduling is a per-iteration process, it operates at a low cost as memory patterns are already discretized into region level (the scheduling overhead will be discussed in Section 7.4). In reality, the slowdown for each training is very predictable when two training jobs exhibit very similar GPU compute demands. A rare but challenging case is when a beefy model is trained along with a lean model which suddenly suffers from a dramatic slowdown from co-location. This case results in memory not being released from the lean model for a long time while the beefy model keeps allocating memory, leading to the system-wide

¹This profiling can also be done offline to reduce online profiling cost.

memory usage going over GPU memory capacity. In this case, Zico gives up an attempt on spatial GPU sharing and falls back to time-multiplexing.

Deadlock potential. With concurrent jobs in phases of increasing memory allocation, the deadlock could happen when there is insufficient memory to allocate to these in-flight jobs. Scrooge scheduler does not start the current iteration of a training job if global memory consumption would go beyond the memory budget during its forward-backward passes. In the worst case, scrooge scheduler will schedule the current iteration when the co-located job finishes its iteration (releasing all memory), guaranteeing training progress similar to temporal sharing and thus preventing the deadlock. Such planned execution should be accompanied by tracking of memory used by GPU, which we will explain in the next section.

6 Memory Management with Concurrency

We discuss how to track GPU memory usage for sharing, classify tensors according to usage type, and manage GPU memory for tensors of different types using regions.

6.1 Tracking Memory Usage in GPU

Most of existing DNN platforms are mainly customized for single-job training and unaware of memory sharing among concurrent training jobs. As described in Section 3.3, inherent asynchrony between CPU and GPU does not incur any correctness issue for memory operations when there is only one job to run on the platform. However, for concurrent training with each job assigned with its own GPU stream, we should not make memory released by CPU readily available for the co-located jobs, since kernel computations to be launched on the multiple GPU streams are independent and unordered. In essence, the memory corruption could occur because there is no dependency among kernels in different GPU streams. We apply two techniques to support efficient memory sharing without the corruption.

Memory deallocation. For memory deallocation, rather than immediately releasing the memory based on CPU point of view, we must wait until GPU actually completes the corresponding kernel execution. Zico facilitates *CUDA event* to meet this requirement. *CUDA event* is a specific kernel that can be inserted into the GPU stream and monitored by CPU for its completion. Once GPU reaches a *CUDA event*, it is guaranteed that kernels launched before the detected *CUDA event* have finished the execution. Hence, tensors of those kernels can be safely released if no longer accessed. Tracking in-use memory with *CUDA event* is not cost-free, requiring CPU cycles. To reduce the overhead, Zico inserts *CUDA event* periodically at a certain memory allocation granularity (currently, 8 MB for CNNs and 256 MB for RNNs). We discuss the sensitivity of the granularity in Section 7.4.

Memory allocation. The memory tracking brings out another issue, namely, *early memory allocation*. Although deallocation of memory occurs after the actual kernel completion as a result of the memory tracking, its allocation occurs by CPU when the kernel is issued to the GPU stream. Thus, the allocation time is typically earlier than the time the kernel actually starts its execution, accesses the memory, and completes the execution within GPU. It is always desirable to maintain a small number of in-flight kernels (i.e., kernels pending on GPU stream), since this way would narrow the above time gap and ultimately avoid unnecessary pre-allocations of memory by CPU: otherwise, the system makes memory allocations too early compared to the actual time the memory is utilized by GPU kernels. Oftentimes, we found memory consumption *soars up* due to a number of in-flight kernels issued at full CPU speed. To address this issue, Zico limits the number of in-flight kernels to a certain level just sufficient to keep GPU busy. Currently, the right number is obtained via offline profiling for each model while not causing a loss in overall training performance. It can be also easily found online by running a few iterations over varying numbers of in-flight kernels to be limited in the GPU.

6.2 Tensor Classification

With a computation graph constructed for training, Zico differentiates the tensors used by GPU kernel operators forming the graph. In general, we classify the current tensors into three types: feature map tensors, gradient map tensors, and temporary tensors, where temporary tensors are neither feature maps nor gradient maps. A temporary tensor is mostly created by an operator and used internally, not later accessed by other operators. To correctly classify tensors in this way, we exploit three pieces of information available for a tensor: by considering the operator creating the tensor as the *source* operator, (i) whether the source belongs to the forward pass; (ii) whether there is any *destination* operator accessing the tensor outside the source; and (iii) whether the destination belongs to the backward pass. Feature map tensors will meet all three conditions while gradient map tensors are distinguished from complying with (ii) and (iii) only. The remaining tensors are all sorted into temporary tensors.

For the proposed tensor classification, we need to identify whether a computation kernel is involved in the forward pass or the backward pass. Memory manager in Zico pinpoints the peak memory as the starting point of the backward pass. This method exploits the fundamental property of DNN training that a forward pass is a memory allocation phase and a backward pass is a memory deallocation phase. This method is a simple but generic approach that does not depend on DNN implementation and does not belong to a specific framework.

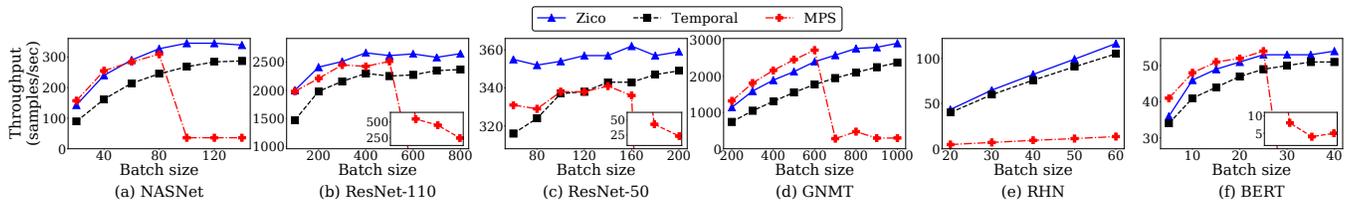


Figure 5: Throughput in training the same models. ResNet-50 and BERT are run on Machine 1 and NASNet, ResNet-110, GNMT, and RHN are on Machine 2 considering their model sizes.

6.3 Managing Memory Regions

Based on the tensor classification, the memory manager in Zico accepts the tensor type as parameter and then allocates the tensor on a region according to the type. The region-based memory management is a basic mechanism in TensorFlow and we extend it to build our own memory sharing system.

The essential goal of Zico’s memory management policy is to promote spatial sharing with low interference between co-scheduled training jobs. Based on separating sharable regions (global) from non-sharable regions (job-local), we assure no contention occurs when non-sharable regions are enough to allocate new tensors for the local job. Further, within the local regions, temporary tensors are stored exclusively on a few regions managed by their own free block lists which manage empty memory space for allocating new tensors. Temporary tensors are small in size and frequently allocated and soon deallocated, thus contending with other types of tensors for accessing free block lists unless managed separately.

The size of the sharable and non-sharable area changes elastically depending on runtime demands. As a region stores tensors in the same type, for feature map tensors the demand will increase and decrease within each iteration. Thus, during forward pass, the local memory manager will continuously request regions from the sharable area and then during the backward pass, these regions will be returned back to the sharable area. The free regions in the sharable area are shared through a free list for which updates are synchronized by a lock. To prevent the potential contention on the lock, the granularity of the regions needs to be chosen carefully. We experimentally validated different region sizes over diverse training jobs. From the sensitivity study, we chose the region size to be at least tens of MB to minimize lock contention. The results of the sensitivity study can be found in Section 7.4.

7 Evaluation

Experimental setup. We implement Zico in TensorFlow 1.13.1 and compare it with spatial sharing using MPS (MPS)² and temporal sharing with no job switching overhead (Temporal), which is similar to the approach taken

²Spatial sharing within a single framework exhibits similar limitations to MPS, i.e., performance degradation with memory oversubscription.

in the state-of-the-art [44]. We select six training benchmarks across different DNN tasks including NASNet [48], ResNet-110 [16], ResNet-50 [16], GNMT [42], RHN [47], and BERT [11]. All models use the stochastic gradient descent (SGD) optimizer.

The evaluation is performed on two machines. Machine1 has an NVIDIA Tesla V100 GPU with 32 GB GPU memory, 3.8 GHz Intel Xeon(R) Gold 5222 4 CPU cores and 64 GB of host memory. Machine2 has an NVIDIA RTX 2080 Ti GPU with 11 GB GPU memory, 3.8 GHz Intel Xeon(R) Gold 5222 4 CPU cores and 64 GB of host memory. Both machines run Ubuntu 16.04. We use Machine1 and Machine2 to evaluate large models and small models, respectively.

7.1 Training Same Models

We first compare Zico, MPS, and Temporal when two identical models are concurrently trained. The memory budget in Zico is configured as GPU memory capacity.

Figure 5 shows the throughput of the six models when training over different input batch sizes (i.e., number of samples) in x-axis. For each model, some of the batch sizes are chosen to have MPS exceed GPU memory capacity to show how effective Zico is in such cases. The figure shows that as compared to temporal sharing, Zico achieves higher throughput across all batch sizes in all models. In particular, Zico outperforms Temporal by on average 35% for NASNet and 37% for GNMT across the batch sizes. These results are rather surprising as the largest batch size in each model results in memory consumption in solo training that reaches close to the GPU memory limit. Even in such an extreme memory usage scenario, Zico finds an optimal time point to start the forward pass of a job while the backward pass of the other job is in progress. Therefore, Zico does not have any model being completely time-multiplexed, making it co-schedule the jobs more efficiently than Temporal.

Zico achieves comparable throughput with MPS from small to modest batch sizes for each model. MPS is sometimes slightly better than Zico. This is not because MPS provides a better schedule for concurrent training but mainly because the underlying setup is different: Zico runs on a single framework and MPS runs two training jobs on different framework instances and processes. Nonetheless, throughput in MPS drops significantly when models are trained on large batch sizes.

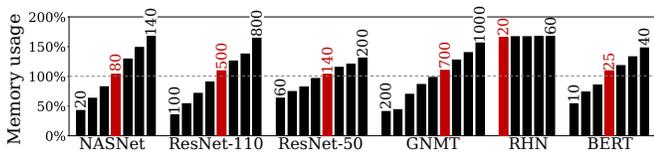


Figure 6: Aggregated memory usage for training the same models. For each model, bars are sorted by the batch sizes as used in Figure 5. The red bars indicate the batch sizes of aggregated memory usage beyond GPU memory capacity.

On training large batches, MPS suffers from GPU memory oversubscription that accompanies UVM overhead. Subsequently, as compared to MPS, Zico is up to 4.7 times faster across models. Note that the solo training of RHN incurs high memory usage even with small batch sizes, causing memory oversubscription for MPS across all batch sizes. Figure 6 presents the system-side memory usage (which sums up the memory usage of the two co-located jobs) to reveal the degree of GPU memory oversubscription handled by Zico.

In Figure 7, we show how memory usage patterns are coordinated in Zico to reduce the system-wide memory footprint. For the space reason, we select only two models, ResNet-110 and BERT, for which concurrent training is scheduled slightly differently. In ResNet-110, almost no delay on each iteration occurs, i.e., $TimeShift \approx 0$, making it behave similar to the non-coordinated spatial GPU sharing. On the contrary, in each scheduling interval of BERT, there is a slight delay applied to every iteration to keep memory consumption within the budget. It is worth noting that for training the same models, the memory-aware schedules across iterations for a job are very regular, making scheduling decisions across the co-located jobs rather deterministic. We also found out training the same models entails an almost similar, if not the same, slowdown for each job. Hence, scrooge scheduler can quickly stabilize its memory-aware scheduling across jobs even without beginning from a low memory budget during the informed phase.

In general, Zico delivers more benefit to less computation-intensive models such as GNMT. Over GPU generations, GPU compute capacity scales faster than GPU memory capacity does, keeping the bottleneck pushed towards GPU memory capacity. With this trend continued, the advantage of Zico over temporal sharing is likely to grow in the future.

7.2 Training Non-identical Models

Now, we use two distinct models in concurrent training. In this experiment, we select five models to make combinations based on different GPU compute demands: GNMT (low), RHN (low), NASNet (high), ResNet-110 (high), and BERT (high). Figure 8 shows the throughput normalized by Temporal over diverse co-location combinations which all oversubscribe GPU memory capacity. In the figure, we put the memory demand of individual training in the parenthesis, computed as the percentage of GPU memory capacity, which

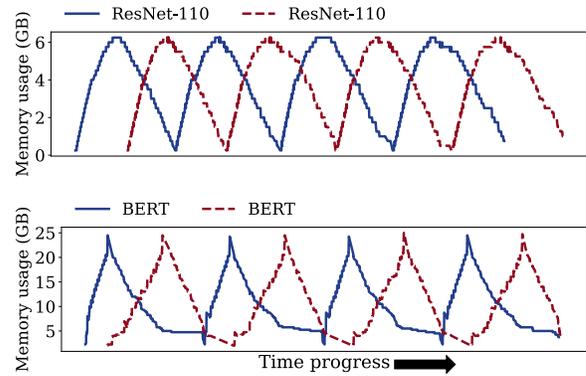


Figure 7: Memory usage over time for training the same models. ResNet-110 and BERT are shown as an example.

is obtained by varying batch sizes for the model.

For training non-identical models, we organize co-location combinations into three scenarios: (i) including both models with low GPU utilization (i.e., RH+GT), (ii) including one model with low GPU utilization (i.e., NN+GT and BT+GT), and (iii) including both models with high GPU utilization (i.e., NN+RN). First, Figure 8 shows that Zico significantly outperforms MPS regardless of GPU utilization between the co-located jobs. Zico is around 5.7x faster than MPS on average, specifically faster up to 5.1x in RH+GT, 8.3x in NN+GT, 6x in BT+GT, and 6.5x in NN+RN. Our conclusion repeats: MPS experiences significant performance degradation under GPU memory oversubscription.

In comparison to Temporal, Zico achieves higher throughput by 42% in RH+GT, 46% in NN+GT, 27% in BT+GT, and 15% in NN+RN on average. That is, Zico favors scenario (i) and (ii) over (iii) because ample GPU cycles are available by running a model with low compute demand. Nonetheless, in Zico, since any of co-located jobs starts training once memory constraint is met, no fair use of compute resources is guaranteed. As a result, in the NN(30%)+GT(90%), Zico obtains a great throughput improvement for GT over Temporal but slightly worse performance for NN due to a bit fewer iterations scheduled within a time duration as opposed to Temporal. We leave balancing individual job throughput on top of increasing the aggregated throughput as future work.

It is also worth noting that Zico achieves different scheduling ratio for the co-located jobs depending on their memory usage patterns. To illustrate, we show memory usage patterns in RN+NN in Figure 9, where ResNet-110 has relatively shorter iteration time compared to NASNet. On the given memory budget, Zico decides to schedule executions of the two jobs in such a way that ResNet-110 runs its iterations more frequently than NASNet does — Zico keeps scheduling ResNet-110 during time periods with low-to-moderate memory usages in NASNet to maximize GPU memory utilization.

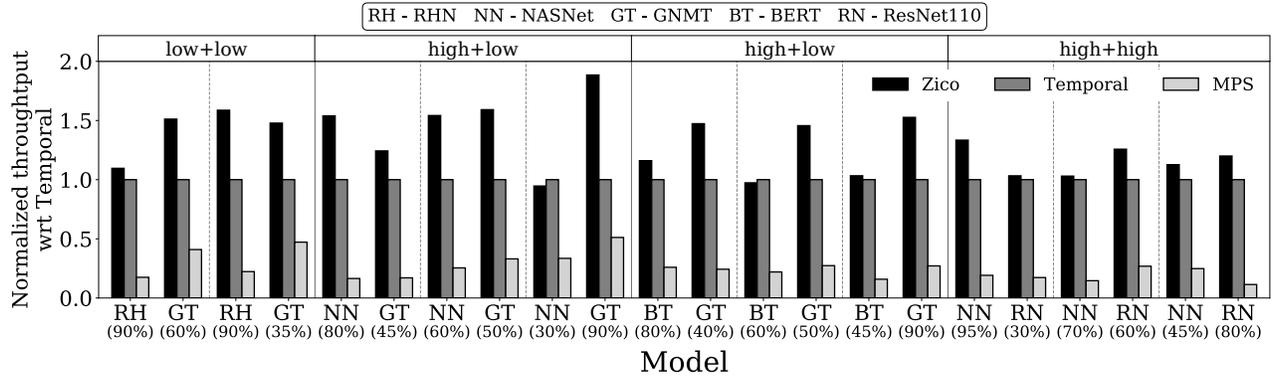


Figure 8: Throughput in training the distinct models concurrently. All co-locations are run on Machine 2.

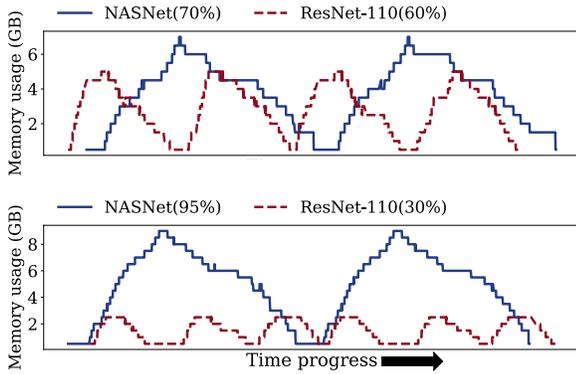


Figure 9: Memory usage over time for training NASNet and ResNet-110 concurrently. This co-location incurs different scheduling ratios for having different memory demands.

7.3 Dynamic Memory Budget Change

Recall that for co-locating more than two jobs, e.g., four jobs, we propose to organize the jobs into a group of pairs and schedule each pair independently with a lower memory budget. Then, when some of the jobs depart, the system would have fewer pairs and need to make a schedule based on the increased memory budget. Therefore, adapting to the memory budget change is a fundamental functionality required in Zico to deal with the dynamic workload. In this section, we evaluate Zico scheduling decisions when several continuous changes are made on the memory budget.

Figure 10 shows how Zico schedules two NASNet training jobs while decreasing the memory budget and then increasing it back. Before the first change, the two jobs run concurrently with zero delay by virtue of scrooge scheduler between consecutive iteration executions of a job under the budget set as 70% GPU memory. Around A time point, the memory budget is set down to 50% and Zico begins to schedule the co-located jobs more conservatively. During this period, each job exhibits a wider gap (140 ms) between consecutive iterations. Around B time point, the memory budget returns back to 70% and Zico now takes a more aggressive schedule on memory sharing. At this moment, we attempt to further increase the

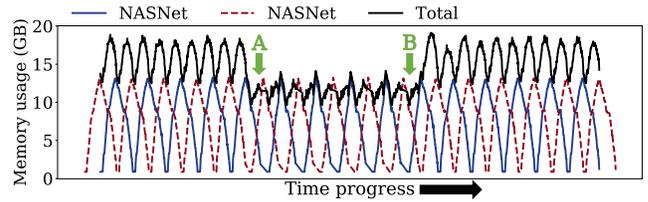


Figure 10: Memory usage patterns on dynamic memory budget changes. The budget is decreased from 70% GPU memory to 50% at time A and then increased back to 70% at time B.

memory budget to fully utilize GPU memory, but we do not see any change in both throughput and memory footprint. The reason is that 70% GPU memory is already enough to make an ideal scheduling decision for Zico with no TimeShift. That is, Zico does not overuse GPU memory unnecessarily.

7.4 Design Validation

GPU memory tracking. As explained in Section 6.1, tracking in-use memory is essential to share memory with the correctness between multiple GPU streams. Table 1 shows the sensitivity of memory tracking granularity. If CUDA event is inserted too frequently, it imposes overhead on CPU and leads to delaying training iteration. For instance, inserting CUDA event for every GPU kernel launch slows down the training up to more than 50% in GNMT. To mitigate this overhead, CUDA event is inserted periodically in Zico. For models which use CPU intensively like GNMT, due to the significant number of light-weight kernels to issue, coarse-grained tracking is required to avoid this overhead. On the

GPU tracking	NN	RN-110	RN-50	GNMT	RHN	BERT
All	78%	91%	97%	44%	94%	97%
Fine-grained	100%	100%	100%	89%	99%	99%
Coarse-grained	100%	100%	100%	97%	99%	100%

Table 1: Throughput with memory tracking (normalized to the throughput with no memory tracking). All, Fine-grained, and Coarse-grained track GPU memory for every kernel launch, 8 MB allocation, and 256 MB allocation, respectively.

Granularity	NN	RN-110	RN-50	GNMT	RHN	BERT
Tensor	94%	99%	98%	90%	96%	94%
Small region	94%	99%	100%	97%	99%	99%

Table 2: Throughput with different sharing granularity (normalized to sharing granularity of 64 MB region size).

other hand, for models which use CPU less like CNN models and BERT, even fine-grained tracking does not bring out the actual delay of the training iteration. Zico chooses the right memory tracking granularity, minimizing the overhead.

Sharing granularity. As mentioned in Section 6.3, if the sharing granularity is too fine-grained, e.g., tensor granularity, the contention of shared lock becomes non-negligible. Table 2 shows the sensitivity of different sharing granularity choices, where the size of small region is set to 512 KB. The table presents the normalized throughput with respect to using our default region size (64 MB) for memory sharing. The tensor-level sharing could introduce up to 10% of throughput degradation as shown in GNMT.

Scheduling. Making a scheduling decision in our scrooge scheduler takes $O(n)$ time complexity, where n is a small number of regions exercised by co-located jobs. The observed overhead is only a few hundreds of nanoseconds, and hence scrooge scheduler has nearly zero scheduling overhead. Moreover, the scheduling process of one job does not interfere with the scheduling process of counterpart co-located job, since each job has a dedicated CPU thread.

8 Related Work

Temporal/Spatial GPU sharing. Temporal GPU sharing represents software-based techniques that time-share GPU for DNN workloads. Gandiva [43] proposes a GPU time-slicing mechanism for the first time to mainly accelerate hyperparameter tuning jobs. It initiates job switching at iteration boundary to reduce CPU-GPU communication overhead. Salus [44] tries to remove the switching overhead by making model parameters of a job resident in GPU memory even when the job is inactive. It further integrates a spatial sharing mechanism to harness underutilized memory in a similar way to MPS [10]. We faithfully compare Zico with both temporal and spatial sharing in Section 7.

Tensor swapping/recomputation. Prior works make use of host memory as a swap storage for DNN training to mitigate memory footprint in GPU [17,32,36]. vDNN [36] predictively swaps tensors ahead of time to overlap CPU-GPU communication with GPU computation during training. It mainly focuses on swapping the inputs of convolutional layers as they tend to have long lifespans in CNN models. SwapAdvisor [17] considers memory allocation and operator scheduling to jointly optimize for swapping decisions. Capuchin [32] proposes a computational graph agnostic technique that estimates

the costs of tensor swapping and recomputation to make the best choice between the two.

Other prior works study dropping tensors created in forward pass and recomputing them in backward pass [6, 19, 41]. SuperNeurons [41] introduces a cost-aware recomputation technique to remove tensors from convolution layer, which are cheap to recompute. Checkmate [19] formulates tensor recomputation into an optimization problem and provides an approximation algorithm to recompute tensors timely. Similar to tensor swapping, tensor recomputation reduces memory footprint for a single training. The goal of Zico is different; Zico reduces global memory footprint for concurrent training.

Compression. Many approaches were invented to reduce memory footprint of DNN training, including HW-based compression techniques [7, 30]. There are also a few SW-based memory compression techniques. Gist [18] proposes a series of layer-specific encoding techniques to compress tensors including feature maps. Echo [46] proposes a compression technique more effective on LSTM RNN training driven by internal operator dependencies. Zico is complementary and can be combined with tensor compression techniques.

9 Concluding Remarks

We present our attempt on realizing GPU memory sharing across concurrent training. The proposed system, Zico, is the first introducing a memory-aware scheduler that coordinates training iterations among co-located jobs with minimum stall times. Zico works generally for co-locating both identical models or non-identical models regardless of the iteration time and the memory pattern of each model. Our experimental results show that Zico outperforms existing GPU sharing approaches. With growing model sizes, very large models such as GPT family [33,34,38] are preferred to run with model parallelism or data parallelism to accommodate intermediate tensors on GPU memory. Despite diverse parallelism in use, we believe Zico benefits both cases as we still see increasing and decreasing memory usage within an iteration.

Acknowledgements

We thank our shepherd, Nandita Vijaykumar, and anonymous reviewers for their valuable comments and suggestions. We also thank Peifeng Yu and Chanho Park for their knowledge sharing and technical support during this work. This work was supported by Samsung Advanced Institute of Technology, the 2021 Research Fund (1.210050.01) of UNIST(Ulsan National Institute of Science and Technology), Electronics and Telecommunications Research Institute(ETRI) grant funded by the Korean government (21ZS1300), and the National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIT) (No. 2020R1C1C1014940 and NRF-2019R1C1C1005166).

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 *USENIX OSDI*, 2016.
- [2] Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, et al. TensorFlow Eager: A Multi-Stage, Python-Embedded DSL for Machine Learning. *arXiv preprint arXiv:1903.01855*, 2019.
- [3] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. In *NeurIPS*, 2011.
- [4] James Bergstra, Daniel Yamins, and David Cox. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *ICML*, 2013.
- [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, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training Deep Nets with Sublinear Memory Cost. *arXiv preprint arXiv:1604.06174*, 2016.
- [7] Esha Choukse, Michael B Sullivan, Mike O’Connor, Mattan Erez, Jeff Pool, David Nellans, and Stephen W Keckler. Buddy Compression: Enabling Larger Memory for Deep Learning and HPC Workloads on GPUs. In *ISCA*, 2020.
- [8] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv preprint arXiv:1412.3555*, 2014.
- [9] NVIDIA Corp. CUDA, release: 10.2.89. <https://developer.nvidia.com/cuda-toolkit>, 2020.
- [10] NVIDIA Corp. Multi-process service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2020.
- [11] 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.
- [12] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to Forget: Continual Prediction with LSTM. *Neural Computation*, 12(10), October 2000.
- [13] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv preprint arXiv:1706.02677*, 2017.
- [14] Andreas Griewank and Andrea Walther. Algorithm 799: Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.
- [15] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *USENIX NSDI*, 2019.
- [16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *CVPR*, 2016.
- [17] Chien-Chin Huang, Gu Jin, and Jinyang Li. SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *ASPLOS*, 2020.
- [18] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient Data Encoding for Deep Neural Network Training. In *ISCA*, 2018.
- [19] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *MLSys*, 2020.
- [20] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, unjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *USENIX ATC*, 2019.
- [21] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *MM*, 2014.
- [22] 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 Snellham, 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 *ISCA*, 2017.
- [23] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NeurIPS*, 2012.
- [25] 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 OSDI*, 2014.
- [26] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and Efficient GPU Cluster Scheduling. In *USENIX NSDI*, 2020.
- [27] Avner May, Alireza Bagheri Garakani, Zhiyun Lu, Dong Guo, Kuan Liu, Aurélien Bellet, Linxi Fan, Michael Collins, Daniel Hsu, Brian Kingsbury, et al. Kernel Approximation Methods for Speech Recognition. *arXiv preprint arXiv:1701.03577*, 2017.
- [28] Tomáš Míkolov, Martin Karafiát, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. Recurrent Neural Network Based Language Model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.
- [29] NVIDIA Corp. NVIDIA A100 Tensor Core GPU Architecture. <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf>, 2020.
- [30] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Bruce Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks. In *ISCA*, 2017.
- [31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *arXiv preprint arXiv:1912.01703*, 2019.

- [32] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-Based GPU Memory Management for Deep Learning. In *ASPLOS*, 2020.
- [33] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving Language Understanding by Generative Pre-Training. 2018.
- [34] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. *OpenAI blog*, 1(8):9, 2019.
- [35] Jeff Rasley, Yuxiong He, Feng Yan, Olatunji Ruwase, and Rodrigo Fonseca. HyperDrive: Exploring Hyperparameters with POP Scheduling. In *Middleware*, 2017.
- [36] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-efficient Neural Network Design. In *MICRO*, 2016.
- [37] Alex Sergeev and Mike Del Balso. Meet Horovod: Uber’s Open Source Distributed Deep Learning Framework for TensorFlow. <https://eng.uber.com/horovod/>, 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] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the Importance of Initialization and Momentum in Deep Learning. In *ICML*, 2013.
- [40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *NeurIPS*, 2017.
- [41] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *PPoPP*, 2018.
- [42] 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.
- [43] 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 *USENIX OSDI*, 2018.
- [44] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications. In *MLSys*, 2020.
- [45] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *USENIX ATC*, 2017.
- [46] Bojian Zheng, Nandita Vijaykumar, and Gennady Pekhimenko. Echo: Compiler-Based GPU Memory Footprint Reduction for LSTM RNN Training. In *ISCA*, 2020.
- [47] Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutnik, and Jürgen Schmidhuber. Recurrent Highway Networks. In *ICML*, 2017.
- [48] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning Transferable Architectures for Scalable Image Recognition. In *CVPR*, 2018.

Refurbish Your Training Data: Reusing Partially Augmented Samples for Faster Deep Neural Network Training

Gyewon Lee^{1,3} Irene Lee² Hyeonmin Ha¹ Kyunggeun Lee¹
Hwarim Hyun¹ Ahnjae Shin^{1,3} Byung-Gon Chun^{1,3*}
Seoul National University¹ Georgia Institute of Technology² FriendliAI³

Abstract

Data augmentation is a widely adopted technique for improving the generalization of deep learning models. It provides additional diversity to the training samples by applying random transformations. Although it is useful, data augmentation often suffers from heavy CPU overhead, which can degrade the training speed. To solve this problem, we propose data refurbishing, a novel sample reuse mechanism that accelerates deep neural network training while preserving model generalization. Instead of considering data augmentation as a black-box operation, data refurbishing splits it into the partial and final augmentation. It reuses partially augmented samples to reduce CPU computation while further transforming them with the final augmentation to preserve the sample diversity obtained by data augmentation. We design and implement a new data loading system, Revamper, to realize data refurbishing. It maximizes the overlap between CPU and deep learning accelerators by keeping the CPU processing time of each training step constant. Our evaluation shows that Revamper can accelerate the training of computer vision models by $1.03\times$ – $2.04\times$ while maintaining comparable accuracy.

1 Introduction

Deep learning (DL) is at the heart of modern AI-based applications, enabling various services such as computer vision [19, 20, 33, 34], automatic speech recognition [38], and natural language processing [17, 35]. DL models are trained by repeatedly adjusting the parameters of the models in order to minimize their loss with regard to training samples. The trained models must ensure generalization, the ability to appropriately process previously unseen input data.

Recently, *data augmentation* has been widely used in deep neural network (DNN) training to improve generalization of DL models including image classification [14, 15, 36, 39], object detection [15, 36], and automatic speech recognition [29]. By applying several transformations on training samples in

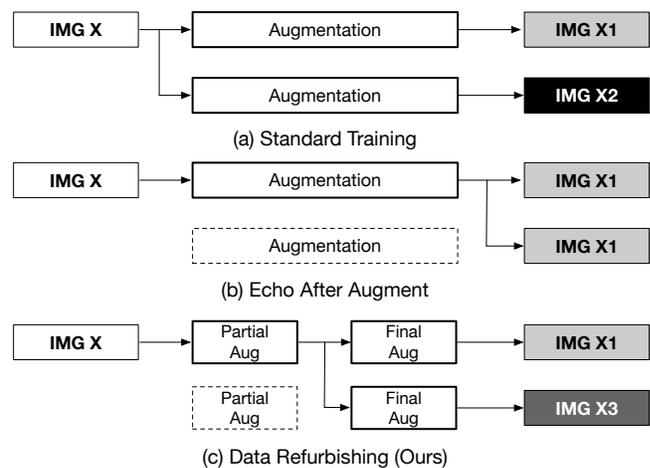


Figure 1: A high-level introduction of (a) standard training, (b) data echoing [9, 13] after augment, and (c) data refurbishing (ours). The dotted rectangles indicate computation saved by sample reuse.

a random manner [14, 15, 29], data augmentation provides additional samples to model training and thus helps improve model generalization. Because data augmentation is a stochastic process, every augmented sample is unique (Figure 1 (a)). Better model generalization from data augmentation, however, comes at the cost of expensive CPU operations. This CPU overhead often causes data augmentation to become a performance bottleneck of DNN training [4, 9, 13, 30].

To address the CPU overhead from data augmentation, recent works such as NVIDIA DALI [4] and TrainBox [30] utilize hardware accelerators such as GPUs and FPGAs for optimizing data augmentation. However, the stochastic nature of data augmentation makes it difficult to exploit accelerators that are optimized for parallel execution of homogeneous operations. Data echoing [9, 13], on the other hand, tries to reduce the amount of computation by splitting training pipelines into the upstream and downstream pipelines, and reusing previously prepared samples from the upstream pipeline in the

*Corresponding author.

downstream pipeline. However, it considers augmentation as a black-box operation and splits the DNN training pipeline to only before or after the augmentation. If the pipeline is split before data augmentation, the overhead from data augmentation remains unchanged. However, with the other option, the augmented samples are reused multiple times without further transformations as shown in Figure 1 (b). This decreases the number of unique samples generated from data augmentation—the sample diversity—to a great degree and degrades the accuracy of trained models.

To solve this problem, we propose data refurbishing, a novel sample reuse mechanism for fast DNN training data preparation. Data refurbishing splits the original data augmentation pipeline into the *partial augmentation* and *final augmentation* according to the given split policy, and reuses the intermediate results generated from the partial augmentation (Figure 1(c)). The *partially augmented samples* produced from the partial augmentation are cached, reused for a designated number of times, and renewed to preserve the diversity of augmented samples. The final augmentation applies the remaining portion of the full augmentation pipeline to the cached samples and produces fully augmented samples to be used for gradient computation.

Although reused, each partially augmented sample undergoes the final augmentation to produce a diverse set of augmented samples to be used for gradient computation. The number of unique samples, thus, remains almost the same, preserving the original sample diversity and model generalization. As such, data refurbishing is able to reduce CPU overhead from data augmentation with little generalization loss. Reducing computation overhead while maintaining enough sample diversity, this approach provides better trade-offs between training speed and model generalization than data echoing. We demonstrate these benefits both mathematically and empirically. Such characteristics make data refurbishing especially useful for exploration tasks, such as hyperparameter optimization [10, 27], which requires much DNN training with various configurations.

We design Revamper, a new caching and data loading system, to realize data refurbishing. Revamper shares similarity with systems that adopt intermediate data caching [18, 37], but it differs from such systems in that Revamper addresses new challenges that are specific to the context of DNN training. Because with data refurbishing a mixture of cached and non-cached samples is used for gradient computation, the CPU processing time may fluctuate depending on the number of cache misses in each step, which can deteriorate computation overlap between the CPU and DL accelerators. Revamper maintains a constant CPU computation overhead across epochs with the balanced eviction and within each epoch with the cache-aware shuffle, where an epoch denotes a complete pass on the entire dataset. The balanced eviction resolves the inter-epoch computation skew by evicting cached samples in a way that the number of cache misses is evenly distributed

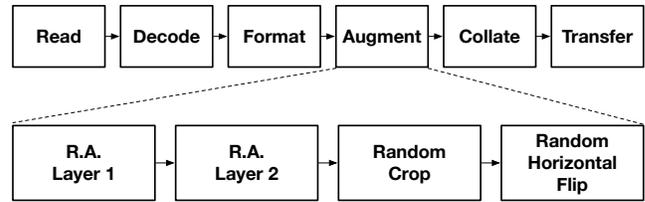


Figure 2: An illustration of a RandAugment [15] augmentation pipeline in a typical data preparation pipeline. R.A. Layer stands for a RandAugment layer.

across epochs. To address the intra-epoch computation skew, the cache-aware shuffle utilizes information from the cache store to prepare training samples for each step to make the number of cache misses uniform over training steps.

Revamper is implemented on PyTorch 1.6 [31], a widely used DL framework for DNN training. Its interface overrides the existing PyTorch dataloader so that users can utilize Revamper by giving a few additional parameters such as how to split the given augmentation pipeline and how many times to reuse each cached sample. Our evaluations on various computer vision models and datasets demonstrate that Revamper can reduce training time for DNN models by $1.03\times$ – $2.04\times$ while maintaining comparable top-1 validation accuracy. Although we focused on evaluating Revamper on vision tasks where data augmentation is most widely used, it is notable that Revamper can also be applied to other domains such as speech [29] and language tasks [32], when augmentation pipelines can be split into multiple transformations.

2 Background

2.1 DNN Training

DNN training "trains" a DL model to perform a certain task by repeatedly adjusting the model weights with regard to the given set of training samples. Broadly speaking, training DNNs consists of two steps: data preparation and gradient computation. Until a certain termination condition is satisfied (e.g. target validation accuracy is met), the two steps are repeated for multiple epochs.

Data Preparation The data preparation step prepares training data to be fed to the DL model. Figure 2 describes a typical data preparation procedure, which is generally performed on CPU [8, 31]. The process starts with reading in the training data residing on a local or a remote storage in a random order in order to give randomness for each epoch. In general, each training data entry is a tuple of (x_i, y_i) , where x_i represents the training sample at index i (e.g. an image, an audio clip, and a text snippet) and y_i the corresponding target of x_i (e.g. class and original text). The loaded training data are decoded and formatted into tensors, multi-dimensional arrays of numbers used in gradient computation.

Often, training data undergo random transformations called *data augmentation* (§ 2.2) in the next step. This optional step gives greater variation in the training set and helps train a more generalized DL model. The decoded and transformed data are collated into mini-batches before being sent to DL accelerators such as GPU and TPU. This mini-batching is necessary to perform stochastic gradient descent or its variants.

Gradient Computation The gradient computation step actually trains a DL model by adjusting the model parameters with regard to the gradients computed from the training data. This is done via forward computation and backward propagation. Forward computation calculates the loss, or the deviation of the produced result from the target value, for the given mini-batch of training samples. Backward propagation traverses the model in a reverse order and recursively computes the gradient of each layer with respect to the loss. The model parameters are then adjusted proportional to the gradients to minimize the loss.

It is important to note that the data preparation and the gradient computation steps can be overlapped, as they are typically executed on different hardware. Thus, ideally the processing time of data preparation can be hidden by that of gradient computation, and so the former has not been considered to have a significant impact on the speed of DNN training. However, recent development of specialized hardware accelerators [2, 3] has dramatically reduced the processing time required for the gradient computation step. Accordingly, the data preparation step is becoming the bottleneck of DNN training as pointed out in the recent works [9, 13, 30].

2.2 Data Augmentation

During the data preparation step, several random distortions, referred to as transformations, are applied to increase the effective number of training samples and thus to improve the generalization of DL models. These data transformation steps are collectively called data augmentation. Data augmentation is a common technique in many domains of DL, including computer vision [14, 15, 36, 39], automatic speech recognition [29] and natural language processing [12, 32].

A data augmentation pipeline is usually a sequence of multiple transformations. Here, each transformation is referred to as a layer. For example, RandAugment [15] consists of a sequence of RandAugment layers (Figure 2), each of which randomly applies one of the 14 distortions (e.g., shear, rotate, and solarize) to each sample. AutoAugment [14] searches a set of effective transformation sequences before training, and applies a sequence randomly selected from the set in every training step. As an example of an extreme use of data augmentation, Karas et al. [23] deployed at most 18 consecutive transformation layers when training generative adversarial network (GAN) models with limited data.

Multi-layered augmentations can also be seen in other do-

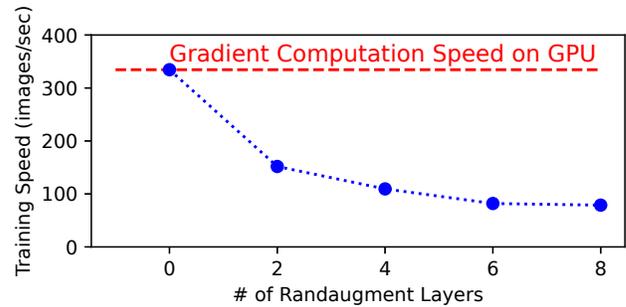


Figure 3: ResNet-50 training speed on ImageNet varying the number of RandAugment layers. The horizontal line indicates the gradient computation speed on GPU.

ains. For example, SpecAugment [29], an augmentation method for automatic speech recognition tasks, can be decomposed into three transformation layers (time warp, frequency masking, and time masking). Such transformations are known to be computationally expensive, which is why popular speech recognition frameworks such as DeepSpeech provides an option that caches and reuses previously augmented samples [7]. Another example is CoSDA-ML [32] in natural language processing, which translates N random tokens into different languages. CoSDA-ML can be decomposed into N consecutive random transformation layers, each of which selects a token and translates it to a token in a randomly chosen language.

3 Motivation

3.1 Overhead of Data Augmentation

Data augmentation improves model generality, but it is often a bottleneck in DNN training due to its heavy CPU overhead from the multiple layers of transformations. To analyze the overhead of data augmentation, we measure the training throughput of ResNet-50 [19] model, a widely-used DL model for image classification, using the ImageNet [16] dataset with an example data preparation pipeline. As for data augmentation, we apply varying number of RandAugment [15] layers along with the random crop and random horizontal flip transformations. The number of RandAugment layers (N) is adjusted from zero to eight to investigate the CPU overhead from various loads of data augmentation. When N is zero, only the random crop and random horizontal flip are applied to training samples. We employ one NVIDIA V100 GPU and four physical CPU cores for the training, which is similar to an Amazon Web Service (AWS) p3.2xlarge instance, a standard cloud virtual machine used for DNN training.

Figure 3 plots the measured throughput of data preparation pipelines with different number of data augmentation layers. When only the random crop and flip are applied ($N = 0$), the

throughput of data preparation exceed that of gradient computation on GPU, making the data preparation step completely overlap with GPU operations. On the other hand, when the number of RandAugment layers is set to 2, which is known to produce the highest validation accuracy when training ResNet-50 on the ImageNet dataset [15], the DNN training process is bottlenecked by the data preparation. This problem becomes more severe as the number of data augmentation increases and the CPU overhead from data augmentation becomes heavier. From the result, we observe that the CPU overhead from data augmentation can be too large to be fully overlapped with the gradient computation, and thus data augmentation can be the main bottleneck in DNN training process.

3.2 Limitations of Existing Approaches

As the data preparation step is becoming the bottleneck for DNN training, there has been effort to reduce this overhead. However, due to the stochastic nature of data augmentation, such effort has failed to efficiently reduce the CPU overhead introduced by data augmentation pipelines.

Utilizing Hardware Accelerators Recent works such as NVIDIA DALI [4] and TrainBox [30] leverage hardware accelerators like GPUs and FPGAs for data augmentation. Unlike the gradient computation, which applies identical and deterministic computations to each training sample, data augmentation applies stochastic operations to each sample in a random fashion. Hence, it is difficult for data augmentation to efficiently utilize such hardware accelerators, which are optimized for massive parallel execution of homogeneous operations [9]. In addition, because such accelerators are frequently used for gradient computation, this approach may make the data preparation and gradient computation not overlapped.

Data Echoing Data echoing [9, 13] splits DNN training pipelines into the upstream and downstream pipelines, and reuses previously produced samples from the upstream pipeline in the downstream. For example, if we split the pipeline in Figure 2 between Format and Augment operations, the upstream pipeline would be Read-Decode-Format and the downstream pipeline would be Augment-Collate-Transfer. This approach is useful when the deterministic part of the data preparation pipeline, such as I/O from a remote storage, is the bottleneck. However, data echoing becomes less effective when stochastic data augmentation is the slowest part. With data echoing, the entire data augmentation pipeline is considered as a black-box operation, and so the samples are reused either before or after the augmentation process. If a sample is reused before the data augmentation, the reused sample needs to be re-augmented, and thus the overhead from data augmentation remains the same. Or, when fully augmented samples are simply reused for gradient computation, the number of unique augmented samples significantly decreases. As a result, data echoing fails

to reduce the CPU overhead from data augmentation without severely harming the generalization of trained models. We further demonstrate this limitation in § 7.

Our observation suggests that it is necessary to devise a mechanism to reduce the computation overhead of CPU-heavy data augmentation techniques, while preserving generalization of the model obtained by data augmentation.

4 Data Refurbishing

We propose **data refurbishing**, a simple and effective sample reuse mechanism for input pipelines of DNN training that alleviates CPU computation for data augmentation while preserving the generalization of trained models. Data Refurbishing caches and reuses partially augmented samples generated from the *partial augmentation*, which consists of the first few transformations in the full augmentation pipeline. The rest of the augmentation pipeline—the *final augmentation*—is applied to the partially augmented samples from the cache in order to produce fully augmented samples. Reusing partially augmented samples reduce CPU computation while further transforming them with the final augmentation maintains the sample diversity obtained by data augmentation.

Data refurbishing introduces two additional configurations, the *reuse factor* and the *split policy*. The reuse factor represents how many times to reuse each cached sample, and the split policy determines how to split the full augmentation pipeline into the partial and final augmentations. Note that configuring Revamper is simple since its configuration space is small. The reuse factor is an integer that is typically smaller than five, and the number of split strategies, which is identical to the number of augmentation layers, does not exceed twenty even in extreme cases [23]. Applying data augmentation without reusing data—the *standard data augmentation*—and data echoing are both special cases of data refurbishing, as will be explained later in this section.

Data refurbishing can reduce the CPU computation required for data augmentation with minimal loss of the generalization of the trained model, given that the final augmentation provides enough sample diversity. In the rest of this section, we mathematically explain how data refurbishing preserves the sample diversity produced from the standard data augmentation.

Problem Formulation Let \mathcal{X} and \mathcal{X}' denote the sample space before and after augmentation, respectively. An augmentation A can then be represented as a finite set of functions such that $A := \{f_1, f_2, \dots, f_{|A|}\}$ for $\forall_i f_i: \mathcal{X} \rightarrow \mathcal{X}'$. Note that, in the standard data augmentation, we randomly choose $f_i \in A$ and produce an augmented sample $f_i(x)$ for a given input sample $x \in \mathcal{X}$ in every epoch. Then, the partial augmentation A_P and the final augmentation A_F of A can also be represented as some augmentations that satisfy $\{f_F \circ f_P | f_P \in A_P, f_F \in A_F\} = A$. In the rest of this section, we make the following assumptions to simplify our analysis.

Assumption 1. Discrete Uniform Distribution

For an augmentation A , the probability of choosing $f \in A$ is uniform.

$$\forall f \in A \ P(f) = \frac{1}{|A|}$$

Assumption 2. Balanced Eviction

All the cached samples are reused exactly r times before being evicted from the cache.

Assumption 3. Uniqueness of Composed Augmentation

Any composition of partial and final augmentation functions always produces a unique fully augmented sample.

$$\begin{aligned} & \forall f_P, g_P \in A_P \ \forall f_F, g_F \in A_F \ \forall x \in \mathcal{X} \\ & ((f_P \neq g_P) \text{ or } (f_F \neq g_F)) \rightarrow (f_F \circ f_P)(x) \neq (g_F \circ g_P)(x) \end{aligned}$$

Now let $A(x)$ denote the set of all possible augmented samples produced by an augmentation A given an input sample x . Then, Assumption 3 implies that $A(x) = \{f_1(x), f_2(x), \dots, f_{|A|}(x)\}$ and $|A_P| \times |A_F| = |A| = |A(x)|$.

Under the above formulation, applying data refurbishing for k epochs with reuse factor r to an augmentation A for an input sample $x \in \mathcal{X}$ can be represented as a sampling process such that the samples are taken r times from $A_F(y)$ for every y sampled $\frac{k}{r}$ times from $A_P(x)$, respectively. Given k , data refurbishing can have any $1 \leq r \leq k$ by the definition. Note that data echoing and the standard data augmentation are both the special cases of data refurbishing, since data echoing is equivalent to data refurbishing with $((A_F = \{I\})$ and $(r > 1))$, and the standard data augmentation is equivalent to data refurbishing with $((A_F = A)$ or $(r = 1))$, where I denotes the identity function.

Therefore, the following theorem holds:

Theorem 1. Expectation of Unique Samples

$$\mathbb{E}(U) = |A| \left(1 - \left(1 - \frac{|A_F|}{|A|} + \frac{|A_F|}{|A|} \left(1 - \frac{1}{|A_F|} \right)^r \right)^{\frac{k}{r}} \right)$$

where U denotes the number of unique samples produced by applying data refurbishing to A given a single input sample.

The proof is given in the supplemental material.

In Theorem 1, $\mathbb{E}(U)$ is maximized when the standard data augmentation ($r = 1$ or $|A_F| = |A|$) is applied, and minimized when data echoing ($r > 1$ and $|A_F| = 1$) is applied. The expected number of unique samples of data refurbishing lies between the two.

Figure 4 visualizes the impact of r and $\frac{\log|A_F|}{\log|A|}$ to $\frac{\mathbb{E}(U)}{\mathbb{E}(U^*)}$ where $\mathbb{E}(U^*)$ denotes the expected number of unique samples of the standard data augmentation. In this figure, we assumed the same data augmentation pipeline used in our evaluation (§ 7), which consists of two RandAugment layers followed by a random crop and a random horizontal flip layers. Intuitively, $\log|A|$ means the number of transformations comprising the

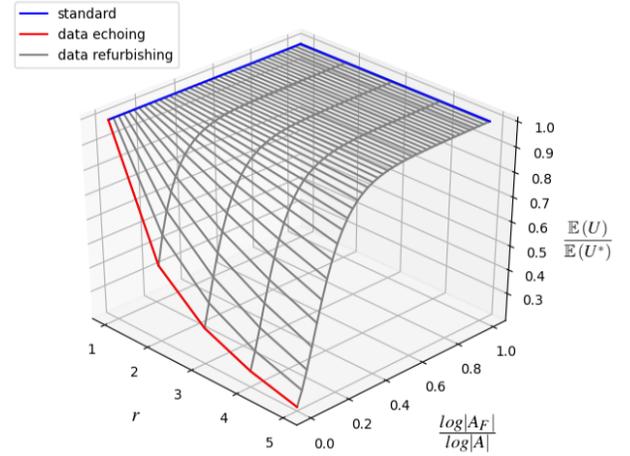


Figure 4: The normalized expected number of unique samples. The x-axis represents reuse factor r , y-axis $\frac{\log|A_F|}{\log|A|}$, and z-axis the normalized expected number of unique samples with respect to that of the standard data augmentation, $\mathbb{E}(U^*)$. Intuitively, the y-axis means the ratio of the number of transformations of final augmentation to that of full augmentation pipeline.

data augmentation pipeline, since $|A|$ grows exponentially as the number of transformations increases. Therefore, the y-axis means the ratio of the number of transformations of final augmentation to that of full augmentation pipeline, assuming each transformation can produce the same number of unique outputs from a given input.

As the figure shows, data refurbishing is robust to the growth of reuse factor r given that $\frac{\log|A_F|}{\log|A|}$ is greater than 0.4. However, when $\frac{\log|A_F|}{\log|A|}$ decreases below 0.4, the expected number of unique samples decreases sharply as r grows. This suggests that we can save computation without significant loss of the model generalization as long as the final augmentation provides sufficient sample diversity.

Based on the above analysis, the goal of a good split policy is to find a split where the final augmentation provides sufficient sample diversity above some threshold with the minimal amount of computation. To do so, it is most desirable for the final augmentation to consist of the transformations that provide high sample diversity with little computation. In our evaluation setup in § 7, for example, one RandAugment layer is computationally heavy but can produce only 14 possible augmented samples from an input sample; on the other hand, random crop ($padding = 3$) along with random horizontal flip can produce a total of 98 augmented samples from an input sample with fewer CPU cycles.

However, if this property does not hold (i.e., the last few layers do not provide sufficient diversity or are computationally heavy), achieving both fast training speed and high accuracy

with data refurbishing might be difficult. In this case, one can consider reordering transformations inside the augmentation pipeline given that the transformations are interchangeable or such reordering puts negligible effect on augmented samples.

5 Revamper Design

We design Revamper, a new data loading system that efficiently implements data refurbishing. It incorporates data refurbishing to existing data preparation procedures of DL frameworks such as PyTorch [31] and TensorFlow [8] by replacing existing data loading systems such as PyTorch dataloader and tf.data [28].

In traditional data loading systems, all the samples in each epoch and step undergo the same data-preparation pipeline. However, with data refurbishing, a mixture of cached and non-cached samples are used to prepare fully augmented samples for the gradient computation. Because cached samples only need the final augmentation to be further applied whereas non-cached samples require both partial and final augmentation, the amount of computation needed for each step and epoch may fluctuate with the number of cache misses. This then causes fluctuation in the CPU processing time. However, the gradient computation time on DL accelerators is consistent throughout the training process, both with and without data refurbishing. For this reason, the CPU processing time may not be effectively overlapped with the DL accelerator processing time when data refurbishing is implemented in a naïve fashion.

Revamper overcomes this challenge by keeping the number of cache misses constant both across epochs and within each epoch, which effectively makes the CPU processing time for each mini-batch consistent throughout the training. First, the **balanced eviction** strategy evenly distributes the number of cache misses across epochs while ensuring that every cached sample is used for gradient computation for the same number of times. Within an epoch, the **cache-aware shuffle** leverages the cache information to choose training samples for mini-batches in order to keep the CPU computation time constant for each step.

We explain broader contexts where Revamper can be used. Revamper is applicable to both local (i.e., only one DL accelerator is used) and distributed (i.e., multiple DL accelerator or machines are used) training environments, because independent Revamper processes are created for each DL accelerator or machine. However, it assumes that training data is accessible from the local disk of each machine, which requires the size of training data that are assigned to each machine to be smaller than the capacity of its local disks. Hence, Revamper currently does not consider network overhead from fetching training data from a shared cloud storage. For such environments, one can consider using distributed caching systems for DNN training [25] along with Revamper.

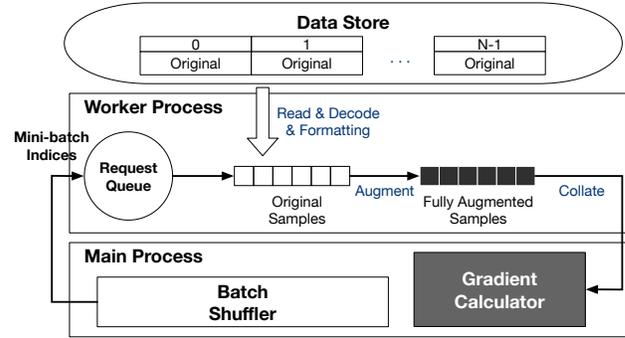


Figure 5: The architecture of a traditional data loading system (PyTorch dataloader).

5.1 Revamper Overview

Before going into Revamper, we briefly explain the existing data loading systems with the architecture of PyTorch dataloader as an example. Figure 5 explains the architecture of PyTorch dataloader, which consists of one main process and one or more worker processes. Each training sample is typically represented with a sequential integer spanning from 0 to $N - 1$, where N denotes the total number of samples. The order of training samples within each epoch is decided by the batch shuffler, which randomly chooses the sample indices for mini-batches. The mini-batch indices are transferred to a worker process. After receiving them, the worker process reads, decodes, and formats the corresponding training samples from the data store. The read samples are then augmented following the user-given augmentation pipeline, making fully augmented samples. The augmented samples are then collated to make a batched sample and transferred to DL accelerators for gradient computation.

The architecture of Revamper (Figure 6) differs from those of traditional data loading systems, mainly in that (1) it has a cache store that stores partially augmented samples in memory or on disk and (2) its main process maintains a separate shuffler that selects the indices to be evicted from the cache. In addition, Revamper adopts the balanced eviction and cache-aware shuffle to stabilize the data preparation time of each mini-batch. Even with such modifications, the epoch boundaries are still intact, meaning that all the original training samples are used exactly once within each epoch of DNN training.

Data Preparation Procedure Figure 6 illustrates the end-to-end data preparation procedure of Revamper in detail. (1) Before starting each training epoch, the evict shuffler selects the samples that need to be evicted from the cache store, following the balanced eviction strategy. By doing so, Revamper balances the number of non-cached samples across epochs while ensuring that each cached sample is reused for the same number of times. (2) The cache store then invalidates the selected samples. (3) After the eviction, the main process

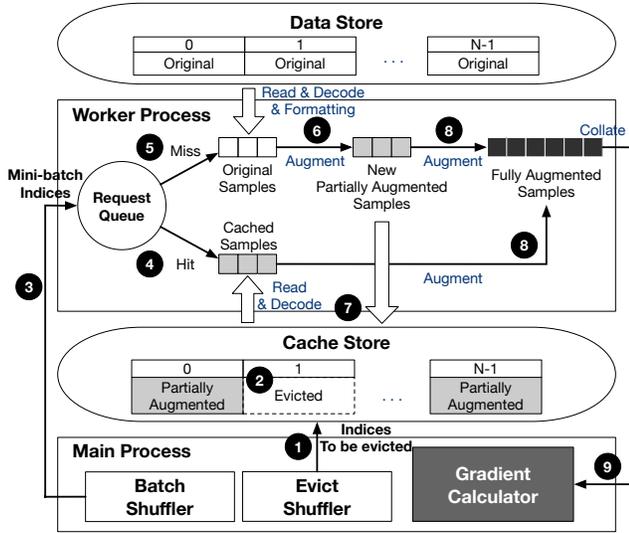


Figure 6: The architecture of Revamper and its end-to-end data preparation procedures.

allocates mini-batch indices to each worker sampled from the batch shuffler. When sampling mini-batch indices, the batch shuffler adopts the cache-aware shuffle in order to make the CPU processing time of each mini-batch stable. (4) Each worker process fetches mini-batch indices from the request queue and reads the corresponding cached samples from the cache store. (5) For the missed samples, the worker process reads the original training samples from the data store and (6) applies the partial augmentation. (7) The worker process then stores the new partially augmented samples in the cache store in order to reuse them in the future epochs. (8) Once all the partially augmented samples for the requested indices are ready, the worker process applies the final augmentation to them. (9) Finally, the fully augmented samples are collated and transferred for the gradient computation.

Cache Store The cache store provides an interface similar to that of key-value stores. It supports `get(I)`, `put(I, S)`, and `remove(I)` methods, where `I` denotes an index and `S` denotes a partially augmented sample to be cached. Partially augmented samples are either stored in memory or on disk according to the user-given `store_disk`. If the `store_disk` is turned off, partially augmented samples are stored in an in-memory hash map that maps indices and the corresponding cached samples. Hash maps are appropriate for DNN training, because it provides $O(1)$ data access by sacrificing performance of range access (i.e., reading all the indices from k_1 to k_2), which is not necessary during DNN training.

Storing partially augmented data on disk is useful when training models with a large dataset whose size exceeds a hundred of GBs [16]. Revamper performs disk I/O in one of the following two ways depending on the size of partially augmented samples. If the size of each sample is below threshold

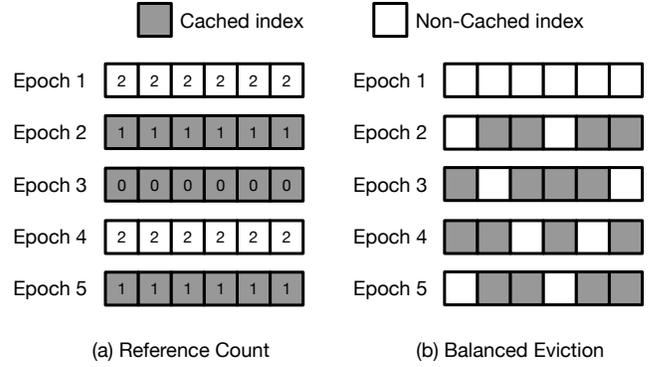


Figure 7: An example distribution of cache misses with (a) reference count algorithm and (b) the balanced eviction.

(16KB by default), Revamper batches multiple I/O requests to reduce system call overhead. To batch write requests, the cached samples are firstly written to in-memory write buffers and flushed to shared log files when the buffers are full. Revamper also batches multiple sample reads within a mini-batch by packing multiple read requests into a single system call using the AIO library of Linux [26]. The cache store periodically clears up invalidated data through background compaction [5], which makes new log files that contain only the valid samples. If the size of each sample is large enough, on the other hand, system call overhead becomes negligible. In this case, Revamper stores each sample in a separate encoded file in order to avoid compaction overhead.

5.2 Balanced Eviction

The balanced eviction is a cache eviction policy that maintains an even spread of computation overhead across the training epochs as well as ensures that each sample is reused for the same number of times. Preparing data from the non-cached samples requires more computation than doing so from the cached samples, because the former needs both the partial and final augmentations to be applied. The computation overhead of each epoch may thus vary depending on the number of cache misses in the epoch when a naïve design of data refurbishing such as the reference count algorithm is used. The reference count algorithm maintains the remaining reference for each cached data and evicts the cached data when a sample’s corresponding reference becomes zero. Although this algorithm ensures that each sample is reused for the same number of times, this approach results in uneven distribution of computation overhead across epochs. As shown in Figure 7 (a), some epochs need to prepare a large number of non-cached samples while others do not, because the remaining references of all the indices decrease at the same rate and become zero at the same time. Such uneven distribution of non-cached samples increases the blocking time between CPU and DL accelerators. Because computation required for

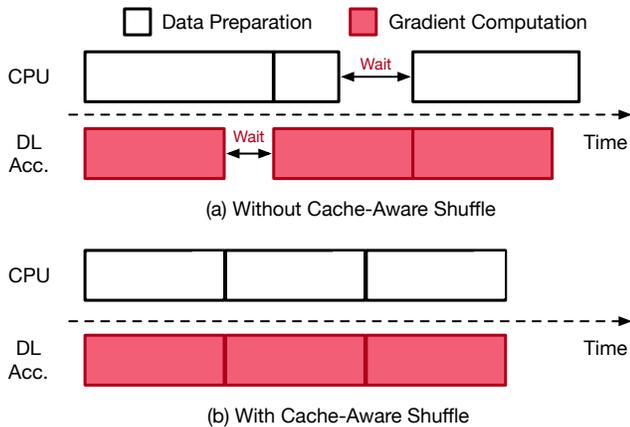


Figure 8: An example illustration of CPU and DL accelerator utilization with and without the cache-aware shuffle. The bidirectional arrow blocking time caused by uneven mini-batch processing time. Each block represents the computation time for corresponding batches.

data augmentation is skewed to a small number of epochs, DL accelerators may wait CPU in such epochs and vice versa in the other epochs.

To solve this problem, we propose the balanced eviction. At the start of each training epoch, the evict shuffler samples $\frac{N}{r}$ indices to be evicted, where N denotes the number of training samples and r denotes the reuse factor. In addition, the shuffler samples the indices without replacement and repeats the same sampling order until the end of training process. By adopting this strategy, the balanced eviction evenly distributes the computation overhead across epochs after the first epoch, by evicting the same number of partially augmented samples in each epoch. It also ensures that except for the the first r epochs, each cached sample is reused exactly r times. Figure 7 (b) illustrates an example of the data loading procedure with $N = 6$ and $r = 3$. After the first epoch, two ($N/r = 2$) samples are evicted from the cache and replaced with new partially augmented samples. All the cached samples are also evicted every three ($r = 3$) epochs, because the evict shuffler always selects indices in a same order.

5.3 Cache-Aware Shuffle

While the balanced eviction effectively addresses the inter-epoch computation skew, the intra-epoch computation skew may still slow down the training speed. Figure 8 (a) illustrates a worst-case example that can happen when the batch shuffler does not adopt the cache-aware shuffle. The non-cached indices are skewed to the first and the third mini-batch, whereas the second batch only contains cached indices. Because the processing time of the data preparation fluctuates while that of the gradient computation remains the same, this results in unnecessary blocking between CPU and DL accelerators.

The cache-aware shuffle solves this problem by leveraging cache information when deciding the indices of the samples for each mini-batch. Because cached samples are evicted only before starting each training epoch, Revamper knows the indices of the evicted samples by the beginning of each epoch. By utilizing this knowledge, the cache-aware shuffle prepares mini-batches in a way that each mini-batch has the same ratio of cached to non-cached samples. Figure 8 (b) shows that the cache-aware shuffle makes the processing time for all mini-batches stable and helps avoid blocking time between CPU and DL accelerators. We ensure the randomness of the mini-batch indices by randomly sampling from both non-cached indices and cached indices. This does not adversely affect the validation accuracy of trained models as shown in § 7.4, because the training order has little impact on the model accuracy as long as it is random [25].

6 Implementation

We implement Revamper with 2000+ lines of code based on PyTorch 1.6 [31] with Python 3.7. Revamper overrides the existing PyTorch dataloader with almost identical interface except for additional parameters such as the reuse factor, the final and partial augmentation, and whether to store cached samples to disk or not. To use Revamper, users only need to override the existing `torch.utils.data` package with our code.

Due to the global interpreter lock (GIL) of Python [11], Revamper workers are executed on separate processes, which makes it hard to share cached samples among the workers if the samples are stored in memory. As a workaround, Revamper puts cached samples inside the main process and transfers them to necessary workers. Such inter-process tensor transfer may cause frequent shared memory allocation, because PyTorch’s `multiprocessing.queue` puts tensors inside the shared memory region when they are transferred between processes. To avoid this problem, Revamper preallocates buffers in the shared memory region and reuses those buffers for inter-process communications.

7 Evaluation

Environment We perform our evaluation on a dedicated training server equipped with 2×Intel Xeon E5-2695v4 CPU (18 cores, 2.10GHz, 45MB Cache), 256GB DRAM, a NVIDIA V100 GPU, and a Samsung 970 Pro 1TB NVMe SSD. We adjust the ratio of the number of CPU cores to the number of GPUs by setting different numbers of CPU cores using a docker container [6] and fixing the number of GPUs to one. By default, we set the CPU-GPU ratio to four, which effectively emulates the ratio in AWS P3 instances [1], but we also evaluate Revamper on different CPU-GPU ratios. Our evaluation is done on PyTorch 1.6. We replace PyTorch

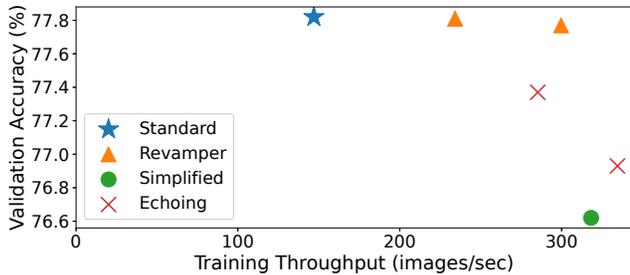


Figure 9: Training throughput and model validation accuracy of ResNet50 trained on ImageNet with diverse settings using RandAugment. Different points of the same setting represent the results under different reuse factors (2 or 3).

dataloader with Revamper for data preparation.

DNN Models and Datasets We evaluate Revamper on several DNN models for image recognition and on two datasets, ImageNet [16] and CIFAR-10 [24], which represent large and small datasets respectively. We train ResNET-50 [19] on ImageNet. On CIFAR-10, we train VGG-16 [33], ResNET-18 [19], MobileNet [20], and EfficientNet-B0 [34].

Baselines We evaluate data refurbishing implemented in Revamper against the following baselines.

- **Standard:** The standard setting represents the canonical DNN training with full augmentation without any reuse mechanism. The accuracy of the model trained under this setting serves as the target accuracy for the other data reusing mechanisms.
- **Data Echoing:** We evaluate data echoing [9, 13] with `echo-after-augment` strategy, in which each fully augmented sample is reused r times, where r denotes the user-given reuse factor. We do not evaluate the other two strategies, `echo-before-augment` and `echo-after-batch`, since they are less relevant and/or not a good baseline. When the training data resides in local SSDs, data echoing with `echo-before-augment` strategy is almost identical to the standard setting with additional encoding and disk write. `echo-after-batch` is reported to result in a lower accuracy with little training throughput improvement [13]. To make a fair comparison, we keep the size of the cache store for data echoing equal to that of Revamper.
- **Simplified:** In this setting DNN models are trained with no reuse mechanism but with fewer transformation layers compared to those of the standard setting. This approach is a baseline optimization for reducing the computation overhead of data augmentation by simply removing one or more transformations.

We use the identical model hyperparameters (e.g., the number of training epochs, learning rate, batch size, and optimizer) for each setting.

We do not evaluate a baseline without augmentation, since random crop and random flip are considered as the norm for training computer vision models. Such baseline only results in a lower accuracy with little improvement on training throughput compared to the simplified setting, as training models without augmentation has been reported to result in a significantly lower accuracy [30], and the simplified setting already makes training throughput bounded by GPUs.

Augmentation and Split Policy We apply RandAugment [15] and AutoAugment [14], the two state-of-the-art data augmentation techniques, accompanied with the random crop and random horizontal flip. After § 7.2, we use RandAugment for the data augmentation methodology.

In most of the experiments except for § 7.2, we use a single split policy: RandAugment or AutoAugment layers for the partial augmentation, and the rest of augmentation (i.e., random crop and random horizontal flip) for final augmentation.

7.1 Comparison with Baselines

ImageNet Training with RandAugment We first evaluate the training throughput and the top-1 accuracy of ResNet-50 trained on ImageNet dataset with RandAugment using Revamper and the other baselines. We follow the model hyperparameters (or configurations) from [15]. We set the number of RandAugment layers (N) to 2 and the distortion magnitude (M) to 9. Then we have a total of four transformation layers, including random crop and flip layers. We also follow the original paper when configuring other hyperparameters, such as the learning rate per batch size, the optimizer and its configurations, and the number of total epochs. We evaluate Revamper and data echoing with two different reuse factors, 2 and 3. For the simplified setting, we use a simpler data augmentation pipeline consisting of a random crop and a random horizontal flip layers. We execute three runs for each point and report the averaged results.

As shown in Figure 9 (a), when training ResNet-50, Revamper achieves top-1 accuracy comparable to the accuracy of 77.82% under the standard setting with better training throughput. With the reuse factor of 2, 77.81% accuracy is achieved with $1.59\times$ training speed-up. With the reuse factor increased to 3, the training throughput improves by $2.04\times$ while maintaining a comparable accuracy of 77.77%. On the other hand, data echoing fails to achieve comparable validation accuracy, having only 77.37% and 76.93% top-1 accuracy for the reuse factor of 2 and 3, respectively. The result demonstrates that Revamper is beneficial over data echoing in that Revamper can maintain comparable accuracy to that of the standard setting, whereas data echoing cannot avoid accuracy degradation even with the smallest reuse factor 2. For example, Revamper with the reuse factor 3 provides 0.84% validation accuracy improvement compared to data echoing with the same reuse factor.

The result also shows that Revamper provides better trade-

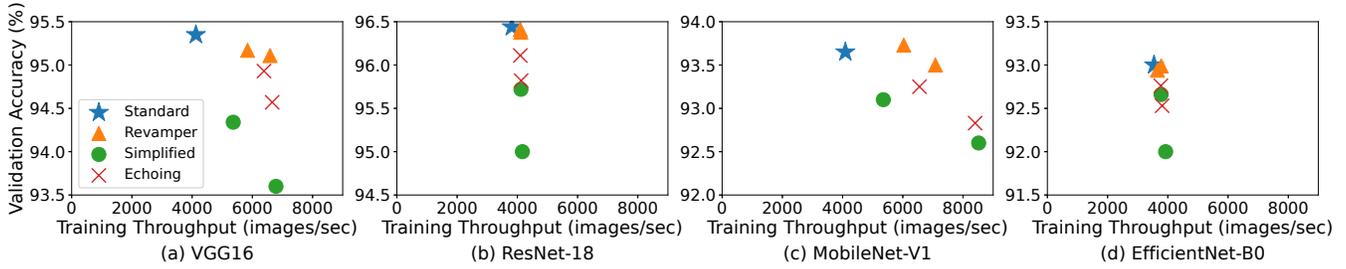


Figure 10: Training throughput and top-1 validation accuracy of DNN models trained on CIFAR-10 using RandAugment. Different points of the same setting represent measurements under different reuse factors (2 or 3) for Revamper and data echoing and under different numbers of removed transformation layers (1 or 2) for the simplified setting.

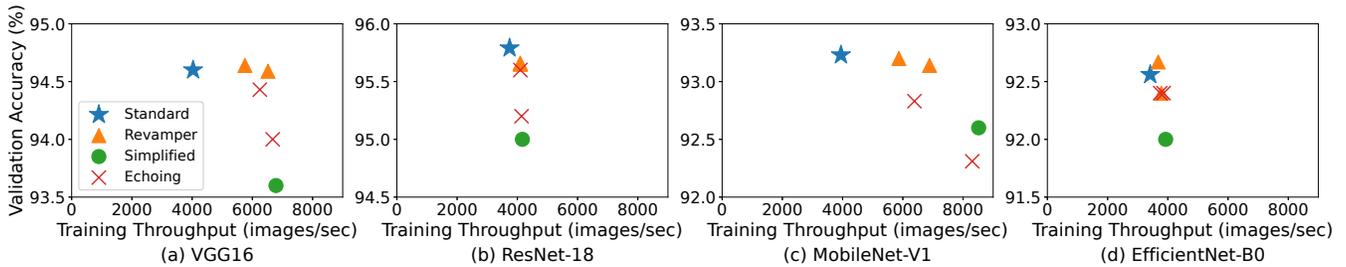


Figure 11: Training throughput and top-1 validation accuracy of DNN models trained on CIFAR-10 using AutoAugment. Different points of the same setting represent the results under different reuse factors (2 or 3).

off points between training throughput and accuracy than data echoing. For example in ResNet-50, compared to data echoing with the reuse factor 2, Revamper with the reuse factor 3 provides faster training throughput ($299.66images/sec$ vs. $285.25images/sec$) and better validation accuracy ($77.77%$ vs. $77.37%$).

The simplified setting achieves the worst validation accuracy with the training throughput similar to that of Revamper with the reuse factor 3, demonstrating that naïvely removing transformations from the pipeline does not provide a good trade-off between training throughput and accuracy.

CIFAR-10 Training with RandAugment Next, we present the performance of our evaluation of training models on a small dataset. We set the number of RandAugment layers (N) to 2 and the distortion magnitude (M) to 10. Same as ImageNet training, we evaluate data refurbishing and data echoing with two reuse factors, 2 and 3. For the simplified setting, we evaluate two different augmentations: one with random crop and random horizontal flip and the other with an additional RandAugment layer. We execute three runs for each point and report the averaged results.

The evaluation results are summarized in Figure 10. For VGG-16 (Figure 10 (a)) and MobileNet-V1 (Figure 10 (c)), Revamper achieves $1.42\times-1.73\times$ speed-up while maintaining validation accuracy comparable to the standard setting. However, for ResNet-18 (Figure 10 (b)) and EfficientNet-B0 (Figure 10 (d)), Revamper does not show significant train-

ing throughput improvement, exhibiting only $1.03\times-1.08\times$ speed-up. This is because these models require more GPU computation time for the gradient computation, and so the training process is bottlenecked by the GPUs rather than the CPUs. Such results suggest that Revamper is beneficial only when DNN training tasks are CPU-bound, but we predict that more training tasks will benefit from Revamper in near future considering rapid performance improvement of DL accelerators [2, 3].

Although data echoing and the simplified setting show an improved training throughput, their validation accuracy deviates much from that of the standard setting. Figure 10 also shows that at many points Revamper demonstrates both higher accuracy and faster training throughput compared to those of data echoing and the simplified setting. Likewise, Revamper provides better trade-offs between training throughput and accuracy than the other baselines.

CIFAR-10 Training with AutoAugment We then evaluate DNN models trained with AutoAugment [14] on CIFAR-10. We use the same configuration we used in the RandAugment evaluation except for the augmentation method. For the simplified setting, we evaluate a data augmentation pipeline with random crop and random horizontal flip, since we cannot manually adjust the number of layers once the policy is found by AutoAugment. We execute three runs for each point and report the averaged results.

Figure 11 demonstrates the results. Compared to the stan-

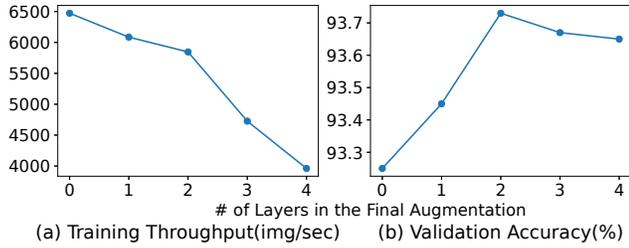


Figure 12: The training throughput and the top-1 validation accuracy for different split policies (MobileNet-V1 on CIFAR-10).

standard setting, Revamper shows $1.08\times-1.75\times$ speed-up while achieving comparable top-1 accuracy within $\pm 0.16\%$ range. Similar to the RandAugment evaluation, models that require less GPU computation (VGG16, MobileNet) has shown greater training throughput gain compared to the models (ResNet-18, EfficientNet-B0) that need heavy GPU computation. Revamper again has better trade-off points between training throughput and accuracy compared to data echoing and the simplified setting.

7.2 Augmentation Split Policy

We evaluate and analyze the trade-offs between training throughput and accuracy for different split policies. The final augmentation contains the last one to three transformation layers of the RandAugment pipeline from Figure 2. Figure 12 summarizes how the number of final augmentation layers affects training throughput and accuracy. Training throughput decreases as the number of the transformations in the final augmentation increases due to heavier CPU overhead. On the other hand, the top-1 accuracy peaks when the final augmentation consists of two transformations and does not increase with additional transformations. This is because the final augmentation with two transformations provides enough sample diversity. As we describe in § 4, once enough sample diversity has been achieved, further providing sample diversity does not significantly improve the model generalization but only degrades the training throughput.

7.3 CPU-GPU Ratio

We evaluate the training throughput change of training MobileNet-V1 on the CIFAR-10 dataset and ResNet50 on the ImageNet dataset with CPU-GPU ratios varying from two to six. As summarized in Figure 13, the training throughput of Revamper scales well upon the increasing number of CPUs, as long as it is not bottlenecked by DL accelerators. Also, the performance gain from Revamper is maximized in training environments with fewer CPUs.

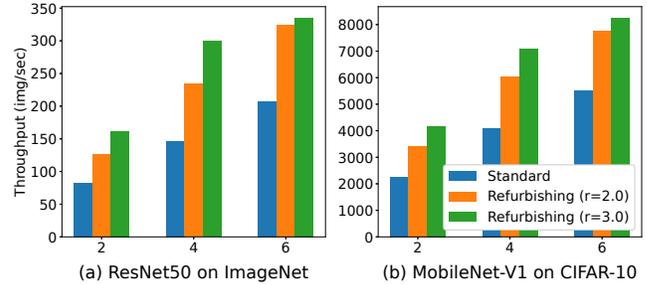


Figure 13: The training throughput of ResNet50 on ImageNet and MobileNet-V1 on CIFAR-10 for varying CPU-GPU ratios.

	Balanced + CAS	Naïve	Standard
VGG-16	5839.08	4746.91	4125.07
ResNet18	4098.36	3884.40	3813.30

Table 1: The comparison of the throughput (*images/sec*) of data refurbishing with and without Revamper’s key features, balanced eviction and cache-aware-shuffle (CAS).

7.4 Revamper Key Design Features

Revamper provides distinctive features—the balanced eviction and cache-aware shuffle—in order to efficiently support data refurbishing. We evaluate how these features further improve the DNN training throughput compared to the naïve approach with the reference count algorithm and the random shuffle. As Table 1 shows, the naïve approach provides $1.15\times$ faster training throughput for VGG-16 training on CIFAR-10 than the standard data loading system, but with the balanced eviction and cache-aware shuffle, the speed-up gain can be as much as $1.42\times$ compared to the standard one. For ResNet18, however, there is no evident additional speed-up gain with the balanced eviction and the cache-aware shuffle. Since the heavy GPU computation needed for ResNet18 training causes the gradient computation to be the main bottleneck, data refurbishing itself has no significant improvement in the training throughput. In summary, the balanced eviction and the cache-aware shuffle of Revamper contributes much to training throughput improvement whenever the DL accelerator is not the main bottleneck. Cache-aware shuffle, although it adjusts the sample order when preparing each mini-batch, does not adversely affect the accuracy of the model, as evidenced in Figure 14. As such, the balanced eviction strategy and cache-aware shuffle help Revamper support data refurbishing efficiently without negatively affecting the model generalization.

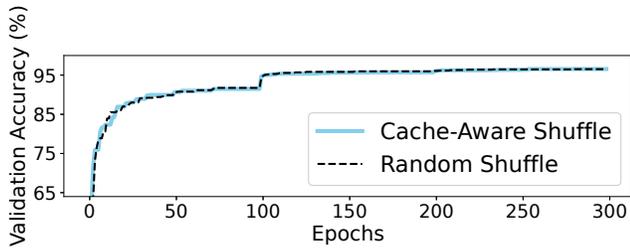


Figure 14: The change in validation accuracy over training epochs of ResNet18 trained on CIFAR-10 with different shuffle strategies.

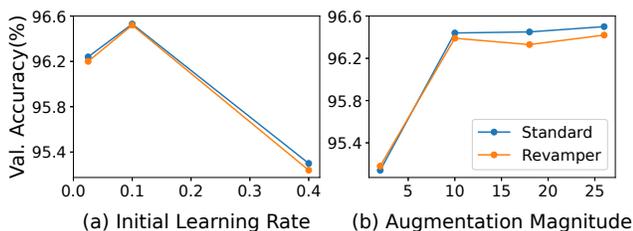


Figure 15: The top-1 validation accuracy of ResNet18 trained with different hyperparameter configurations.

7.5 Robustness to Hyperparameter Change

In this evaluation, we show that Revamper preserves the model accuracy of the standard setting under various hyperparameters. We have varied two hyperparameters—the initial learning rate and the distortion magnitude of RandAugment. As shown in Figure 15, the accuracy of the model trained with Revamper is well aligned with the one trained with the standard setting. This indicates that Revamper can achieve the validation accuracy comparable to that of the standard setting on various hyperparameter configurations.

8 Related Work

We discussed the limitations of existing approaches that accelerate data augmentation in § 3.2. In this section, we introduce other works that are closely related to our system.

Accelerating Data Preparation Quiver [25] proposes a caching system between local and remote storage shared by multiple DNN training jobs, in order to optimize slow data read from remote storage with limited cache. To achieve this goal, it leverages internal information of DL frameworks to optimize cache loading and eviction. Quiver and Revamper differ in that Quiver focuses on sharing cached training data among multiple tasks while ensuring randomness of training order, whereas Revamper focuses on reusing partially augmented data while keeping sample diversity obtained from data augmentation. OneAccess [21] proposes to use a shared data preparation pipeline to train multiple DNN models with

the same dataset. Revamper, on the other hand, focuses on optimizing data augmentation within a single DNN training task. SMOL [22] dynamically adjusts the fidelity of input data to avoid data preparation bottleneck at inference time. Unlike SMOL, Revamper focuses on fast DNN training with data augmentation rather than DNN inference.

Intermediate Data Caching Many data processing systems such as Spark [37] and Nectar [18] adopts intermediate data caching, which caches and reuses frequently used intermediate results in order to reduce computation overhead. Similarly, Revamper reuses intermediate results in the augmentation pipeline but instead handles stochastic data. In the context of DL pipelines, it is necessary to consider new aspects that have not yet been considered in previous work, such as maintaining the diversity of augmented samples and maximizing computation overlap between CPU and DL accelerators. In this work, we propose complete system design and implementation that address these new challenges.

9 Conclusion

In this paper, we present data refurbishing, a novel sample reuse mechanism that accelerates DL training while maintaining model generalization. We realize this idea by designing and implementing Revamper, a new caching and data loading system that solves system-side challenges from caching partially augmented data. Revamper has shown $1.03\times-2.04\times$ speed-up in DNN training while maintaining comparable accuracy. We hope that this work will encourage further research to rethink well-studied topics like caching in systems in the new context of deep learning.

Acknowledgments

We thank our shepherd Jonathan Mace and the anonymous reviewers for their insightful feedback. We also thank Youngseok Yang, Taegeon Um, Soojeong Kim, Taebum Kim, Yunmo Koo, Jaewon Chung, and the other members of the Software Platform Lab at Seoul National University for their comments on the draft. This work was supported by the MSIT(Ministry of Science, ICT), Korea, under the High-Potential Individuals Global Training Program (2020-0-01649) supervised by the IITP(Institute for Information & Communications Technology Planning & Evaluation), the ICT R&D program of MSIT/IITP (No.2017-0-01772, Development of QA systems for Video Story Understanding to pass the Video Turing Test), and Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2015-0-00221, Development of a Unified High-Performance Stack for Diverse Big Data Analytics).

References

- [1] Amazon EC2 P3 - Ideal for Machine Learning and HPC - AWS. <https://aws.amazon.com/ec2/instance-types/p3/>.
- [2] Cloud Tensor Processing Units (TPU). <https://cloud.google.com/tpu/docs/tpus>.
- [3] NVIDIA A100. <https://www.nvidia.com/en-us/data-center/a100/>.
- [4] NVIDIA DALI. <https://github.com/NVIDIA/DALI>.
- [5] RocksDB Compaction. <https://github.com/facebook/rocksdb/wiki/Compaction>.
- [6] Runtime options with Memory, CPUs, and GPUs. https://docs.docker.com/config/containers/resource_constraints/.
- [7] Training Your Own Model—DeepSpeech 0.9.3 documentation. <https://deepspeech.readthedocs.io/en/r0.9/TRAINING.html#augmentation>.
- [8] 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 *USENIX OSDI*, pages 265–283, 2016.
- [9] Naman Agarwal, Rohan Anil, Tomer Koren, Kunal Talwar, and Cyril Zhang. Stochastic optimization with laggard data pipelines. *NeurIPS*, 33, 2020.
- [10] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *ACM SIGKDD*, pages 2623–2631, 2019.
- [11] David Beazley. Understanding the python gil. In *PyCON Python Conference*, 2010.
- [12] Jiaao Chen, Zichao Yang, and Diyi Yang. Mixtext: Linguistically-informed interpolation of hidden space for semi-supervised text classification. *arXiv preprint arXiv:2004.12239*, 2020.
- [13] Dami Choi, Alexandre Passos, Christopher J Shallue, and George E Dahl. Faster neural network training with data echoing. *arXiv preprint arXiv:1907.05550*, 2020.
- [14] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation strategies from data. In *IEEE CVPR*, pages 113–123, 2019.
- [15] Ekin D Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V Le. Randaugment: Practical automated data augmentation with a reduced search space. In *NeurIPS*, 2020.
- [16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *IEEE CVPR*, pages 248–255. Ieee, 2009.
- [17] 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.
- [18] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic management of data and computation in data-centers. In *USENIX OSDI*, volume 10, pages 1–8, 2010.
- [19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE CVPR*, pages 770–778, 2016.
- [20] 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.
- [21] Aarati Kakaraparthi, Abhay Venkatesh, Amar Phani-shayee, and Shivaram Venkataraman. The case for unifying data loading in machine learning clusters. In *USENIX HotCloud*, 2019.
- [22] Daniel Kang, Ankit Mathur, Teja Veeramacheneni, Peter Bailis, and Matei Zaharia. Jointly optimizing preprocessing and inference for dnn-based visual analytics. *VLDB*, 14(2):87–100, 2020.
- [23] Tero Karras, Miika Aittala, Janne Hellsten, Samuli Laine, Jaakko Lehtinen, and Timo Aila. Training generative adversarial networks with limited data. *NeurIPS*, 33, 2020.
- [24] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.
- [25] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *USENIX FAST*, pages 283–296, 2020.
- [26] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *ACM SOSP*, pages 447–461, 2019.

- [27] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-Tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning. In *MLS*, volume 1, 2020.
- [28] Derek G Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf. data: A machine learning data processing framework. *arXiv preprint arXiv:2101.12127*, 2021.
- [29] Daniel S Park, William Chan, Yu Zhang, Chung-Cheng Chiu, Barret Zoph, Ekin D Cubuk, and Quoc V Le. SpecAugment: A simple data augmentation method for automatic speech recognition. *Interspeech*, pages 2613–2617, 2019.
- [30] Pyeongsu Park, Heetaek Jeong, and Jangwoo Kim. Trainbox: An extreme-scale neural network training server architecture by systematically balancing operations. In *IEEE/ACM MICRO*, pages 825–838. IEEE, 2020.
- [31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pages 8026–8037, 2019.
- [32] Libo Qin, Minheng Ni, Yue Zhang, and Wanxiang Che. Cosda-ml: Multi-lingual code-switching data augmentation for zero-shot cross-lingual nlp. In *ICJAI*, pages 3853–3860, 2020.
- [33] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [34] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.
- [35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, pages 5998–6008, 2017.
- [36] Sangdoon Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *IEEE ICCV*, pages 6023–6032, 2019.
- [37] 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 *USENIX NSDI*, pages 15–28, 2012.
- [38] Albert Zeyer, Kazuki Irie, Ralf Schlüter, and Hermann Ney. Improved training of end-to-end attention models for speech recognition. *Interspeech*, pages 7–11, 2018.
- [39] Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. In *ICLR*, 2018.

ZeRO-Offload: Democratizing Billion-Scale Model Training

Jie Ren
UC Merced

Samyam Rajbhandari
Microsoft

Reza Yazdani Aminabadi
Microsoft

Olatunji Ruwase
Microsoft

Shuangyan Yang
UC Merced

Minjia Zhang
Microsoft

Dong Li
UC Merced

Yuxiong He
Microsoft

Abstract

Large-scale model training has been a playing ground for a limited few users, because it often requires complex model refactoring and access to prohibitively expensive GPU clusters. ZeRO-Offload changes the large model training landscape by making large model training accessible to nearly everyone. It can train models with over 13 billion parameters on a single GPU, a 10x increase in size compared to popular framework such as PyTorch, and it does so without requiring any model change from data scientists or sacrificing computational efficiency.

ZeRO-Offload enables large model training by offloading data and compute to CPU. To preserve compute efficiency, it is designed to minimize data movement to/from GPU, and reduce CPU compute time while maximizing memory savings on GPU. As a result, ZeRO-Offload can achieve 40 TFlop-s/GPU on a single NVIDIA V100 GPU for 10B parameter model, compared to 30TF using PyTorch alone for a 1.4B parameter model, the largest that can be trained without running out of memory on GPU. ZeRO-Offload is also designed to scale on multiple-GPUs when available, offering near-linear speedup on up to 128 GPUs. Additionally, it can work together with model parallelism to train models with over 70 billion parameters on a single DGX-2 box, a 4.5x increase in model size compared to using model parallelism alone.

By combining compute and memory efficiency with ease-of-use, ZeRO-Offload democratizes large-scale model training making it accessible to even data scientists with access to just a single GPU.

1 Introduction

Since the advent of the attention-based deep learning (DL) models in 2017, we have seen an exponential growth in DL model size, fueled by substantial quality gains that these attention based models can offer with the increase in the number of parameters. For example, the largest language model in literature had less than 100M parameters in 2017. It grew to over 300M with BERT [6] in 2018, and increased to tens of billions in 2019 with models such as GPT-2 [3], T5 [20], Megatron-LM [28] and Turing-NLG [25]. Today, the largest

language model GPT-3 [2] has a staggering number of 175B parameters. With the three orders of magnitude growth in model size since 2017, the model accuracy continues to improve with the model size [12]. Recent studies in fact show that larger models are more resource-efficient to train than smaller ones [12] for a given accuracy target. As a result, we expect the model size to continue growing in the future.

However, accessibility to large model training is severely limited by the nature of state-of-art system technologies. Those technologies make entry into the large model training space prohibitively expensive. To be more specific, distributed parallel DL training technologies such as pipeline parallelism [10], model parallelism [28], and ZeRO [21] (Zero Redundancy Optimizer) allow transcending the memory boundaries of single GPU/accelerator device by splitting the model states (parameters, gradients and optimizer states) across multiple GPU devices, enabling massive models that would otherwise simply not fit in a single GPU memory. All record-breaking large models such as GPT-2, Megatron-LM, Turing-NLG, and GPT-3, were trained using a combination of the aforementioned technologies. However, all of these DL parallel technologies require having enough GPU devices such that the aggregated GPU memory can hold the model states required for the training. For example, training a 10B parameter model efficiently requires a DGX-2 equivalent node with 16 NVIDIA V100 cards, which costs over 100K, beyond the reach of many data scientists, and even many academic and industrial institutions.

Heterogeneous DL training is a promising approach to reduce GPU memory requirement by exploiting CPU memory. Many efforts have been made in this direction [8, 9, 11, 17, 23, 23, 24, 32–34]. Nearly all of them target CNN based models, where activation memory is the memory bottleneck, and model size is fairly small (less than 500M). However, the primary memory bottleneck for recent attention based large model training are the model states, instead of activation memory. There is an absence in literature studying these workloads for heterogeneous DL training. Additionally, existing efforts on heterogeneous training are further limited in two major ways: i) nearly all of them exploit CPU memory, but not CPU compute, which we show can be used to significantly reduce the CPU-GPU communication overhead, and

ii) they are mostly designed for and evaluated on single GPU [9, 11, 23, 34], without a clear path to scaling efficiently on multiple GPUs, which is crucial for large model training.

Addressing the aforementioned limitation, we attempt to democratize large model training by developing ZeRO-Offload, a novel heterogeneous DL training technology designed specifically for large model training. ZeRO-Offload exploits both CPU memory and compute for offloading, while offering a clear path towards efficiently scaling on multiple GPUs by working with ZeRO-powered data parallelism [21]. Additionally, our first principle analysis shows that ZeRO-Offload provides an optimal and the only optimal solution in maximizing memory saving while minimizing communication overhead and CPU compute overhead for large model training.

ZeRO-Offload is designed around three main pillars: i) Efficiency, ii) Scalability, and iii) Usability.

Efficiency: The offload strategy in ZeRO-Offload is designed with the goal of achieving comparable compute efficiency to the state-of-art non-offload strategies but for significantly larger models. To achieve this goal, we rely on first principle analysis to identify a *unique optimal* computation and data partitioning strategy between CPU and GPU devices. This strategy is optimal in three key aspects: i) it requires orders-of-magnitude fewer computation on CPU compared to GPU, preventing the CPU compute from becoming a performance bottleneck, ii) it minimizes the communication volume between CPU and GPU preventing communication from becoming a bottleneck, and iii) it provably maximizes memory savings on GPU while achieving minimum communication volume.

Our analysis shows that to be optimal in the aforementioned regards, we must offload the gradients, optimizer states and optimizer computation to CPU, while keeping the parameters and forward and backward computation on GPU. This strategy enables a 10x increase in model size, with minimum communication and limited CPU computation, which allows us to train 13B parameters on a single NVIDIA V100 GPU at 40 TFLOPS, compared to 30 TFLOPS on the same GPU with 1.2B parameters, the largest model that can be trained without any CPU offloading.

Offloading optimizer computation requires CPU to perform $O(M)$ computation compared to $O(MB)$ on GPU where M and B are the model size and batch sizes respectively. In most cases, the batch size is large, and CPU computation is not a bottleneck, but for small batch sizes, the CPU compute can be a bottleneck. We address this using two optimizations: i) an efficient *CPU optimizer* that is up to 6x faster than the-state-of-art, and ii) One-step *delayed parameter update* that allows overlapping the CPU optimizer step with GPU compute, while preserving accuracy. Together, they preserve efficiency for ZeRO-Offload even with small batch sizes.

Scalability: Good scalability is crucial to take advantage of multiple GPUs that may be available to some data scientists. In the DL community, data parallelism is generally

```
import torch
...
model = BuildModel(config)
optimizer = Optimizer(model)
...
for batch in batches:
    loss = model(batch)
    loss.backward()
    optimizer.step()

import torch
import deepspeed
...
model = BuildModel(config)
optimizer = Optimizer(model)
model = deepspeed.initialize(
    model, optimizer)
...
for batch in batches:
    loss = model(batch)
    model.backward(loss)
    model.step()
```

Figure 1: ZeRO-Offload can be enabled with a few lines of change. The code on left shows a standard training pipeline, while the right shows the same pipeline with ZeRO-Offload enabled.

used as the de facto standard to scale DL training to multiple GPUs [5, 26, 35]. However, it is not designed to work with heterogeneous training and presents scalability challenges because of the replication of data and computation in data parallel training. Data parallel training replicates all the model states such as optimizer states, parameters, and gradients, and it also replicates the optimizer computation on each GPU. Therefore, offloading model states or optimizer computation to CPU in combination with data parallelism will result in significant replication of communication and CPU compute: increase the CPU memory requirement proportionally to the data parallelism degree while limiting throughput scalability due to the increased communication.

To address these challenges, ZeRO-Offload combines unique optimal offload strategy with ZeRO [21] powered data parallelism instead of traditional data parallelism. The symbiosis allows ZeRO-Offload to maintain a single copy of the optimizer states on the CPU memory regardless of the data parallel degree. Furthermore, it keeps the aggregate communication volume between GPU and CPU, as well as the aggregate CPU computation a constant regardless of data parallelism, allowing ZeRO-Offload to effectively utilize the linear increase in CPU compute with the increase in the data parallelism degree. As a result, ZeRO-Offload achieves excellent scalability on up to 128 GPUs.

In addition to working with ZeRO powered data parallelism, ZeRO-Offload can be combined with model parallelism [27, 28] to achieve higher memory savings, when multiple GPUs are available.

Usability: ZeRO-Offload is available as part of an Open-Source PyTorch library, DeepSpeed (www.deepspeed.ai). Unlike most strategies discussed in Section 2, ZeRO-Offload does not require model refactoring to work. In fact, PyTorch users can enable ZeRO-Offload with few lines of code change to their existing training pipeline as shown in Figure 1, allowing to train 10x larger models easily.

Contributions. To the best of our knowledge, ZeRO-Offload is the first fully distributed all-reduced based training framework using CPU memory and computation resources to train large-scale models. We summarize contributions as follows:

- A unique optimal offload strategy for heterogeneous large model training on GPU + CPU system that enables 10x larger model on a single GPU without sacrificing efficiency (Sec. 3 and Sec. 4.1).
- Highly scalable multi-GPU design through i) a symbiotic combination of offload strategy with ZeRO powered data parallelism (Sec. 4.2), allowing ZeRO-Offload to achieve near-linear scalability, and ii) seamless integration with model-parallel training [28], enabling even larger models than using ZeRO-Offload or model parallelism alone (Sec. 4.2).
- Open-source implementation of ZeRO-Offload in PyTorch.
- Extensive evaluation demonstrating i) *Model Scale*: 10x increase in model size with up to 13B on a single GPU and 4x increase in model size over model parallelism with up to 70B parameters on a DGX-2 node. ii) *Efficiency*: Over 40 TFlops for a 10B parameters on a single NVIDIA V100, compared to 30 TFLOPS on the same GPU with 1.4B parameters, the largest model that can be trained without any CPU offloading; Outperform two state-of-the-art heterogeneous DL training frameworks by 22% and 37% respectively on a single GPU. iii) *Scalability*: Near-perfect linear scalability for a 10B parameter model on up to 128 GPUs. iv) CPU overhead reduction with our ADAM implementation with 6x speedup over PyTorch optimizer and up to 1.5X improvement in end-to-end throughput with delayed parameter update optimizations (Sec. 6).

2 Background and Related Work

Memory consumption in large model training. The full spectrum of memory consumption during DL model training can be classified into two parts: i) model states and ii) residual states [21]. Model states include parameters, gradients, and optimizer states (such as momentum and variances in Adam [13]); Residual states include activations, temporary buffers, and unusable fragmented memory.

Model states are the primary source of memory bottleneck in large model training. We consider the memory consumption due to model states for large transformer models such as Megatron-LM (8 billion) [28], T5 (11 billion) [20], and Turing-NLG [25] (17.2 billion). They are trained with float-16 mixed precision training [16] and Adam optimizer [13].

Mixed precision training often keeps two copies of the parameters, one in float-16 (fp16) and the other in float-32 (fp32). The gradients are stored in fp16. In addition to the parameters and gradients, the Adam optimizer keeps track of the momentum and variance of the gradients. These optimizer states are stored in fp32. Therefore, training a model in mixed precision with the Adam optimizer requires at least 2 bytes of memory for each fp16 parameter and gradient, and 4 byte of memory for each fp32 parameter, and the momentum and variance of each gradient. In total, a model with M parameters requires $16 \times M$ bytes of memory. Therefore, the model states

for Megatron-LM, T5 and Turing-NLG require 128 GB, 176 GB and 284 GB, respectively, which are clearly beyond the memory capacity of even the current flagship NVIDIA A100 GPU with 80 GB of memory.

Significant amount of work has been done in the recent years to enable large model training, which requires more memory than what is available on a single GPU to fit these model and residual states. These efforts can be classified broadly into two categories: i) scale-out training and ii) scale-up training based approaches. We discuss them as follows.

Scale out large model training. Scale-out training uses aggregate memory of multiple GPUs to satisfy the memory requirement for large model training. Two prominent examples of scale out training is model parallelism [5, 28] and pipeline parallelism [7, 10], both partitioning the model states and the residual states across multiple GPUs. Model parallelism [5, 28] partitions the model vertically and distributes the model partitions to multiple GPU devices in order to train large models. Pipeline parallelism [7, 10] on the other hand parallelizes model training by partitioning the model horizontally across layers. Both of these approaches must change the user model to work, therefore can limit usability.

A recent work, ZeRO [21], provides an alternative to model and pipeline parallelisms to train large models. ZeRO splits the training batch across multiple GPUs similar to data parallel training [5, 26, 35], but unlike data parallel training which replicates all the model states on each GPU, ZeRO partitions them across all GPUs, and uses communication collectives to gather individual parameters as needed during the training. ZeRO does not require changes to the user model to work, making it more generic than model or pipeline parallel training. It also offers better compute efficiency and scalability.

Despite the ability of model and pipeline parallelisms, and ZeRO to train large models, they all require multiple GPUs such that the aggregate GPU memory can hold the model and residual states for training large models. In contrast, ZeRO-Offload is designed to fit a larger model by offloading model states to CPU memory and can train a 10x larger model on a single GPU without sacrificing efficiency. When multiple GPUs are available, ZeRO-Offload is designed to work together with ZeRO to offer excellent scalability, or in conjunction with model parallelism to fit even larger model sizes that is not possible with ZeRO-Offload or model parallelism alone.

Scale up large model training. Existing work scales up model size in a single GPU through three major approaches. The first approach trades computation for memory saving from activations (residual memory) by recomputing from checkpoints [4]. The second approach uses compression techniques such as using low or mixed precision [16] for model training, saving on both model states and activations. The third approach uses an external memory such as the CPU memory as an extension of GPU memory to increase memory capacity during training [8, 9, 11, 17, 23, 24, 33].

Our work, ZeRO-Offload falls under the third approach.

Unlike ZeRO-Offload, the above efforts only offload data to CPU but not compute, and they use smaller models training. Furthermore, none of the above works is communication optimal, leading to extra communication between CPU and GPU and hurting training throughput. In contrast, a recent work called L2L [18] can enable multi-billion parameter training by managing memory usage in GPU layer by layer. In particular, L2L synchronously moves tensors needed in the upcoming layer into GPU memory for computation and keeps the rest of tensors into CPU memory for memory saving. In comparison to ZeRO-Offload, it offers limited efficiency due to extra communication overhead, does not offer a way to scale out across devices, and requires model refactoring, making it difficult to use.

ZeRO powered data parallel training. ZeRO-Offload works with ZeRO to scale DL training to multiple GPUs. ZeRO has three stages, ZeRO-1, ZeRO-2 and ZeRO-3 corresponding to the partitioning of the three different model states, optimizer states, gradients and parameters, respectively. ZeRO-1 partitions the optimizer states only, while ZeRO-2 partitions gradients in addition to optimizer states, and ZeRO-3 partitions all model states. ZeRO-Offload works symbiotically with ZeRO-2, and therefore we discuss it further.

In ZeRO-2, each GPU stores a replica of all the parameters, but only updates a mutually exclusive portion of it during the parameter update at the end of each training step. As each GPU only updates a portion of the parameters, they only store optimizer states and gradients required to make that update. After the update, each GPU sends its portion of the updated parameters to all the other GPUs using an `all-gather` communication collective. ZeRO-2 computation and communication schedule is described below:

During the forward pass, each GPU computes the loss with respect to a different mini-batch. During the backward propagation, as each gradient is computed, it is averaged using a `reduce` operator at the GPU/GPUs that owns the gradient or part of the gradient. After the backward pass, each GPU updates its portion of the parameters and optimizer states using the averaged gradients corresponding to that portion. After this, an `all-gather` is performed to receive the rest of the parameter update computed on other GPUs.

3 Unique Optimal Offload Strategy

ZeRO-Offload is designed to enable efficient large model training on a single or multiple GPUs by offloading some of the model states from GPU to CPU memory during training. As discussed in Sec. 2, model states: parameters, gradients, and the optimizer states, are the primary source of memory bottleneck in large model training. By offloading some of these model states to CPU, ZeRO-Offload can enable training of significantly larger models ¹. However, identifying the

¹ZeRO-Offload only offloads model states. Offloading secondary sources of memory bottleneck such as activation memory is beyond scope of our

optimal offloading strategy is non-trivial. There are numerous ways to offload model states to CPU memory, each with a different trade-off in terms of CPU computation, and GPU-CPU communication, both of which can limit the training efficiency.

To identify the optimal offload strategy, ZeRO-Offload models the DL training as data-flow graph and uses first principle analysis to efficiently partition this graph between CPU and GPU devices. ZeRO-Offload partitions the graph in a way that is *optimal* in three key aspects: i) it requires orders-of-magnitude fewer computation on CPU compared to GPU, which prevents CPU from becoming a performance bottleneck (Sec. 3.1), ii) it guarantees the minimization of communication volume between CPU and GPU memory (Sec. 3.3), and iii) it provably maximizes the memory savings while achieving minimum communication volume (Sec. 3.4). In fact, ZeRO-Offload can achieve high efficiency during training that is comparable to non-offload training and it is *unique optimal*, meaning no other solution can offer better memory savings without increasing the communication volume or increasing CPU computation.

In this section, we discuss the derivation of our unique optimal offload strategy. Our strategy is specifically designed for *mixed precision training with Adam optimizer* which is the de facto training recipe for large model training.

3.1 DL Training as a Data-Flow Graph

The DL training workload can be represented as a weighted directed graph of data and computation, as shown in Figure 2, where the circular nodes represents model states (parameter16, gradient16, parameter32, momentum32, variance32), and the rectangular nodes represents computation (forward, backward, param update). The edges in the graph represents the data flow between the nodes, and the weight of an edge is the total data volume in bytes that flows through it during any given training iteration. For a model with M parameters, the weight of the edges in this graph is either $2M$ where the source node produces fp16 model states, or $4M$ where the source node produces fp32 model states.

An offload strategy between GPU and CPU can be represented using a two-way partitioning of this graph, such that computation nodes in a partition would be executed on the device that owns the partition, and the data nodes in the partition will be stored on device that owns the partition. The total data volume that must be communicated between GPU and CPU is given by the weight of edges running across two partitions.

There are numerous ways to partition this graph. In the following sections, we use first principles to simplify the data

offload strategy. Given that they are significantly smaller than model states, we ignore them for the purpose of our analysis. Furthermore, the first and second approaches described in Sec. 2 can be used in conjunction with ZeRO-Offload to reduce activation memory

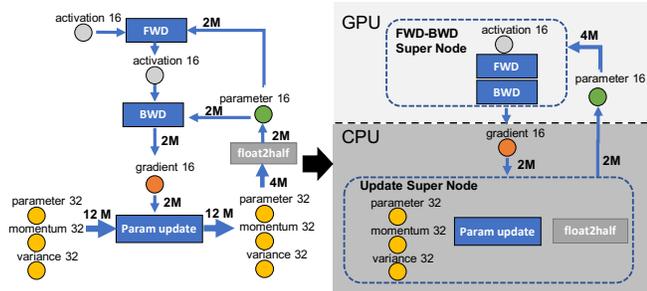


Figure 2: The dataflow of fully connected neural networks with M parameters. We use activation checkpoint to reduce activation memory to avoid activation migration between CPU and GPU.

flow graph to reduce the number of possible choices based on three different efficiency metric: i) CPU computation overhead, ii) communication overhead, and iii) memory savings.

3.2 Limiting CPU Computation

The CPU computation throughput is multiple orders of magnitude slower than the GPU computation throughput. Therefore, offloading large computation graph to CPU will severely limit training efficiency. As such, we must avoid offloading compute intensive components to the CPU.

The compute complexity of DL training per iteration is generally given by $O(MB)$, where M is the model size and B is the effective batch size. To avoid CPU computation from becoming a bottleneck, only those computations that have a compute complexity lower than $O(MB)$ should be offloaded to CPU. This means that the forward propagation and backward propagation both of which have a compute complexity of $O(MB)$ must be done on GPU, while remaining computations such as norm calculations, weight updates etc that have a complexity of $O(M)$ may be offloaded to CPU.

Based on this simple observation we fuse the forward and backward nodes in our data flow graph into a single super-node (FWD-BWD) and assign it to GPU.

3.3 Minimizing Communication Volume

The CPU memory bandwidth is at least an order of magnitude faster than the PCI-E bandwidth between CPU and GPU, while the GPU memory is another order of magnitude faster than even the CPU memory. Therefore, we must minimize the communication volume between CPU and GPU memory to prevent the PCI-E bandwidth from becoming a training performance bottleneck. To do so we must first identify the theoretical minimum communication volume for a model-state offload strategy.

The minimum communication volume for any model-state offload strategy is given by $4M^2$. Note that after fusing the

²Please note that it is possible to reduce the communication volume further by only offloading partial model states. For simplification, we assume

forward and backward into a single super-node as discussed in Sec. 3.2, each node in our data flow graph is part of a cycle. Therefore, any partitioning of this graph would require cutting at least two edges, each of which has a edge weight of at least $2M$, resulting in a total communication of at least $4M$.

If we choose to limit the communication volume to this bare minimum, we can greatly simplify our data-flow graph and reduce the number of partitioning strategies to a handful:

Creating fp32 super-node. Notice that any partitioning strategy that does not co-locate the fp32 model states with their producer and consumer nodes cannot achieve the minimum communication volume of $4M$. Such a partition must cut at least one edge with a weight of $4M$, and the other with at least $2M$, resulting in a communication volume of at least $6M$. Therefore, to achieve the minimum communication volume, all offload strategies must co-locate fp32 model states with their producer and consumer operators, i.e., the fp32 model states (momentum32, variance32 and parameter32) must be co-located with the *Param Update* and the *float2half* computation.

This constraint allows us to treat all the aforementioned fp32 data and compute nodes in the data flow graph as a single super-node that we refer to as *Update Super*. We show this reduced data flow graph in Figure 2, consisting of only four nodes: *FWD-BWD Super* node, *p16* data node, *g16* data node, and *Update Super* node.

p16 assignment. To achieve the minimum communication volume, *p16* must be co-located with *FWD-BWD Super* because the edge weight between these two nodes is $4M$. Separating these two nodes, would increase the communication volume to $6M$ (i.e., $4M + 2M$). Since, we have already assigned node *FWD-BWD Super* to GPU to limit computation on CPU, *p16* must also be assigned to GPU.

3.4 Maximizing Memory Savings

After simplifying the data flow graph to minimize communication volume, only *g16* and *Update Super* remain to be assigned. Notice that at this point, all partitions will result in minimum communication volume, so we can prune the choices further to maximize the memory savings on GPU. Table 1 shows the memory savings of all valid partitioning strategies that minimize the communication volume. The maximum memory savings of 8x can be achieved by offloading both *g16* and *Update Super* to CPU.

Table 1: Memory savings for offload strategies that minimize communication volume compared to the baseline.

FWD-BWD	p16	g16	Update	Memory	Reduction
gpu	gpu	gpu	gpu	16M	1x (baseline)
gpu	gpu	cpu	gpu	14M	1.14x
gpu	gpu	gpu	cpu	4M	4x
gpu	gpu	cpu	cpu	4M	8x

that an offload of a model state implies that we offload the entire model state. Our analysis on the memory savings per communication volume, still holds even if we offload partial model states

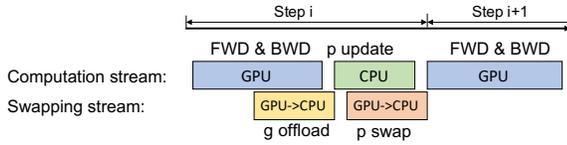


Figure 3: ZeRO-Offload training process on a single GPU.

3.5 A Unique and Optimal Offload Strategy

ZeRO-Offload allocates all the fp32 model states along with the fp16 gradients on the CPU memory, and it also computes the parameter updates on CPU. The fp16 parameters are kept on GPU and the forward and backward computations are also done on GPU.

We arrive at this offload strategy by simplifying our data flow graph and eliminating all other partitioning strategies as they do not limit CPU computation, minimize communication volume, or maximize memory savings. Therefore, ZeRO-Offload is not only optimal in terms of the aforementioned metrics, it is also unique; there can be no other strategy that can offer more memory savings than ZeRO-Offload without increasing the compute complexity on the CPU or incur additional GPU-CPU communication volume.

4 ZeRO-Offload Schedule

In this section, we discuss the concrete computation and communication schedule for implementing ZeRO-Offload on a single GPU system based on our offload strategy. We then show how we extend this schedule to work effectively on multi-GPU systems by combining our offload strategy with ZeRO data parallelism and model parallelism.

4.1 Single GPU Schedule

As discussed in Sec. 3, ZeRO-Offload partitions the data such that the fp16 parameters are stored in GPU while the fp16 gradients, and all the optimizer states such as fp32 momentum, variance and parameters are stored in CPU.

During the training, we begin by computing the loss via the forward propagation. Since the fp16 parameters are already presented on GPU, no CPU communication is required for this part of the computation. During the backward propagation on the loss, the gradient for different parameters are computed at different point in the backward schedule. ZeRO-Offload can transfer these gradients for each parameter individually or in small groups to the CPU memory immediately after they are computed. Therefore, only a small amount of memory is required to temporarily hold the gradients on the GPU memory before they are transferred to CPU memory. Furthermore, each gradient transfer can be overlapped with the backpropagation on the remainder of the backward graph, allowing ZeRO-Offload to hide a significant portion of the communication cost.

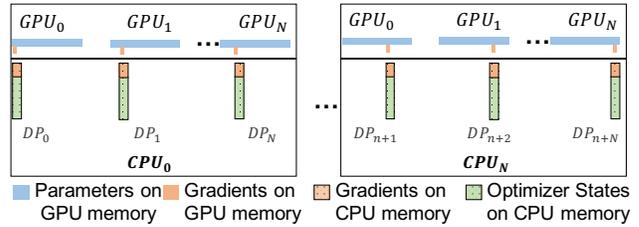


Figure 4: ZeRO-Offload data placement with multiple GPUs

After the backward propagation, ZeRO-Offload updates the fp32 parameters and the remaining optimizer states (such as momentum and variance) directly on CPU, and copies the updated fp32 parameters from the CPU memory to the fp16 parameters on the GPU memory. Figure 3 shows the computation and communication in each step of ZeRO-Offload diagrammatically, and Figure 5 shows the concrete schedule as a pseudo-code.

4.2 Scaling to Multi-GPUs

ZeRO-Offload in its entirety is a symbiotic integration of ZeRO-Offload strategy described in Sec. 3 and ZeRO-powered data parallelism discussed in Sec. 2, which allows ZeRO-Offload to scale to hundreds of GPUs efficiently. ZeRO-Offload preserves the model state partitioning strategy of ZeRO Stage-2 (optimizer state and gradient partitioning), while offloading the partitioned gradients, optimizer states and the corresponding parameter updates to CPU.

The key benefit of doing this partitioning before offloading is that for systems with more than 1 GPU, each data parallel process is only responsible for updating a subset of the parameters. The aggregated communication volume from all the data parallel GPUs to CPU remains constant, and CPU resources are used in parallel to jointly compute a single weight update. As a result, the total CPU update time decreases with increased data parallelism, since the CPU compute resources increase linearly with the increase in the number of compute nodes. This allows ZeRO-Offload to achieve very good scalability, as the overhead of communication across GPUs is offset by the reduction in the CPU optimizer step.

ZeRO-Offload partitions gradients and optimizer states among different GPUs, and each GPU offloads the partition it owns to the CPU memory and keeps it there for the entire training. During the backward propagation, gradients are computed and averaged using reduce-scatter on the GPU, and each GPU only offloads the averaged gradients belonging to its partition to the CPU memory. Once the gradients are available on the CPU, optimizer state partitions are updated in parallel by each data parallel process directly on the CPU. After the update, parameter partitions are moved back to GPU followed by an all-gather operation on the GPU similar to ZeRO-2 to gather all the parameters. Figure 4 shows the data

placement model parameters, gradients and optimizer states for ZeRO-Offload and the details of the ZeRO-Offload data parallel schedule is presented in Figure 5. The all gather operation described above is shown as a sequence of broadcast operations in the Figure.

Model Parallel training ZeRO-Offload can also work together with tensor-slicing based model parallelism (MP) frameworks such as Megatron-LM [28]. It does so by offloading the gradients, optimizer states and the optimizer computation corresponding to each MP process allowing ZeRO-Offload to train significantly larger models than possible than using model parallelism alone. Sec. 6 provides more details.

5 Optimized CPU Execution

We speedup the CPU execution time for the parameter updates with two optimizations. First, we implement a fast CPU Adam optimizer using high performance computing techniques offering significant speedup over state-of-art Pytorch implementation. Second, we develop a one-step delayed parameter update schedule that overlaps the CPU parameter update computation with the forward and backward computation on the GPU, hiding the CPU execution time when enabled.

5.1 Implementing the CPU Optimizer

We use three levels of parallelism for improving the performance of the CPU optimizer. 1) SIMD vector instruction [15] for fully exploiting the hardware parallelism supported on CPU architectures. 2) Loop unrolling [31], an effective technique for increasing instruction level parallelism that is crucial for better memory bandwidth utilization. 3) OMP multithreading for effective utilization of multiple cores and threads on the CPU in parallel. Using these technique, we present a significantly faster implementation of Adam optimizer compared to state-of-art PyTorch implementation.

Mixed Precision Training with Adam ADAM is an optimization algorithm used for deep-learning training, which takes the loss gradients together with their first and second momentums to update the parameters. Therefore, in addition to the model parameters, ADAM requires two more matrices of the same size (M) saved during the training. In the mixed precision training mode, there are two versions of the parameters stored in memory: one in fp16 (parameter16) used for computing the activations in the forward pass (on GPU), and one master copy in fp32 (parameter32) which is updated by the optimizer (on CPU). The p16 is updated with the parameter32 through *float2half* casting, at each training step. Moreover, the momentum and variance of the gradients are saved in fp32 (on CPU), to prevent the precision loss for updating the parameters. Please refer to [13] for further detail on ADAM’s algorithm.

```

1  for_parallel rank in range(world_size):
2      initialize_layers()
3      for batch in dataset:
4          x = forward(batch)
5          compute_loss(x, batch).backward()
6          backward(x.grad)
7          step()
8
9  def _is_owner(i):
10     return True if rank owns i else False
11
12  def initialize_layers():
13     for i in range(num_layers):
14         l = layers[i]
15         allocate_on_gpu l.param_fp16
16         if _is_owner(i):
17             allocate_on_cpu l.param_fp32
18             allocate_on_cpu l.optim_states_fp32
19             allocate_on_cpu l.cpu_grad
20
21  def forward(x):
22     for i in range(num_layers):
23         x = layers[i].forward(x)
24     return x
25
26  def backward(dx):
27     for i in range(num_layers, 0, -1):
28         dx=layers[i].backward(dx)
29         reduce(layers[i].grad, dest_rank
30              = _owner_rank(i))
31         if _is_owner(i) l.cpu_grad.copy(l.grad)
32         else pass
33         del layers[i].grad
34
35  def step():
36     for i in range(num_layers):
37         l=layers[i]
38         if _is_owner(i):
39             update_in_cpu(l.optim_states_fp32,
40                          l.cpu_grad,
41                          l.param_fp32)
42             l.param_fp16.copy(l.param_fp32)
43     BROADCAST(l.param_fp16, src=_owner_rank(i))

```

Figure 5: Code representing ZeRO-Offload that combines unique optimal CPU offload strategy with ZeRO-powered data parallelism.

Optimized Implementation Algorithm 1 elaborates the ADAM’s implementation detail using SIMD operations. As shown, the Adam function receives the optimizer parameters such as β_1 , β_2 , and α , and the gradient, momentum, variance and master copy of parameters (parameter32) as the input. We also use some parameters specific to the implementation, like the *simd_width* and *unroll_width*. The Adam optimizer sends back the updated variance, momentum, and parameter in both fp16 (to GPU) and fp32 (to CPU) .

We firstly read the data, including parameter, gradient, momentum and variance, into the vector registers (line 7). Then, we use several fused multiply-add (*FMA*) vector operations to preform the main execution pipeline which is repeated by the unrolling width. Note that the rest of operations, such as multiply, division, and sqrt, also run in vector mode. For the best performance we use AVX512 simd instruction set and an *unroll_width* of 8 based on auto-tuning results.

In addition to the CPU-Adam optimizer, we implement

Algorithm 2 CPU-ADAM Optimizer

Input: $p32, g32, m32, v32, \beta_1, \beta_2, \alpha, step, eps$ **Output:** $p16, p32, m32, v32$ **Parameter:** $tile_width, simd_width, unroll_width$

```
1:  $biascorrection1 \leftarrow -\alpha / (1 - \beta_1^{step})$ 
2:  $biascorrection2 \leftarrow 1 / \sqrt{1 - \beta_2^{step}}$ 
3:  $simd\_count \leftarrow \text{sizeof}(32) / simd\_width$ 
4: unroll omp parallel
5: for  $i$  in 1 to  $(simd\_count / unroll\_width)$  do
6:   ...
7:    $g_v, p_v, m_v, v_v = g32[i], p32[i], m32[i], v32[i]$ 
8:    $m_v = \text{FMA}(g_v, (1 - \beta_1), \beta_1 * m_v)$ 
9:    $v_v = \text{FMA}(g_v * g_v, (1 - \beta_2), \beta_2 * v_v)$ 
10:   $g_v = \text{FMA}(\sqrt{v_v}, biascorrection2, eps)$ 
11:   $g_v = m_m / g_v$ 
12:   $p_v = \text{FMA}(g_v, biascorrection1, p_v)$ 
13:   $p32[i], m32[i], v32[i] = p_v, m_v, v_v$ 
14:  ...
15:  IF  $(i == tile\_width)$   $\text{copy\_to\_gpu}(p16, p32)$ 
16: end for
```

the CPU-to-GPU fp16 parameter-copy in a tiled manner (line 15). We overlap the CPU and GPU execution by parallelizing the Adam computation and copying the parameters over to GPU. As we process Adam computation of the current tile of data on CPU, we write the parameters back to GPU for the previously processed tile. This way, we reduce the idle time of GPU to start the processing of the next training step.

5.2 One-Step Delayed Parameter Update

Despite using a highly optimized CPU optimizer, the CPU computation overhead can become a bottleneck during training with very small batch sizes, when the GPU computation time is not much larger than CPU compute. For such limited cases, we develop one-step delayed parameter update (DPU) that overlaps CPU and GPU compute to hide the CPU computation overhead by delaying the parameter update by a single step. We verify that DPU does not impact the final accuracy of training in the evaluation.

DPU training schedule Figure 6 shows the workflow of ZeRO-Offload training process with delayed parameter update. ❶ The first $N - 1$ steps, are trained without DPU to avoid destabilizing the training during the early stages where gradients change rapidly. ❷ On step N , we obtain the gradients from the GPU, but we skip the CPU optimizer step, and do not update the fp16 parameters on the GPU either. ❸ At step $N + 1$, we compute the parameter updates on the CPU using gradients from step N , while computing the forward and backward pass on the GPU in parallel using parameters updated at step $N - 1$. From this step onwards, the model at $(i + 1)^{th}$ step will be trained using the parameters updated with gradients from $(i - 1)^{th}$ step instead of parameters updated at i^{th} step, overlapping CPU compute with GPU compute.

Accuracy trade-off. Since DPU changes the semantics of the training, it is reasonable to ask if there is a trade-off between model accuracy and training efficiency. To answer this

Table 2: Hardware overview of experimental system.

DGX-2 node	
GPU	16 NVIDIA Tesla V100 Tensor Core GPUs
GPU Memory	32GB HBM2 on each GPU
CPU	2 Intel Xeon Platinum 8168 Processors
CPU Memory	1.5TB 2666MHz DDR4
CPU cache	L1, L2, and L3 are 32K, 1M, and 33M, respectively
PCIe	bidirectional 32 GBps PCIe

question, we evaluated DPU on multiple training workloads and found that DPU does not hurt convergence if we introduce DPU after a few dozen iterations instead of introducing it at the beginning. Our evaluation result in Sec. 6 shows that compared with training with ZeRO-Offload only, training with delayed parameter update achieves same model training accuracy with higher training throughput.

6 Evaluation

This section seeks to answer the following questions, in comparison to the state-of-the-art:

- (i) How does ZeRO-Offload scale the trainable model size compared to existing multi-billion parameter training solutions on a single GPU/DGX-2 node?
- (ii) What is the training throughput of ZeRO-Offload on single GPU/DGX-2 node?
- (iii) How does the throughput of ZeRO-Offload scale on up to 128 GPUs?
- (iv) What is the impact of our CPU-Adam and delay parameter update (DPU) on improving throughput, and does DPU change model convergence?

6.1 Evaluation Methodology

Testbed. For the evaluation of model scale and throughput, we conduct our experiments on a single DGX-2 node, whose details are shown in Table 2. For the evaluation of throughput scalability, we conduct experiments on 8 Nvidia DGX-2 nodes connected together with InfiniBand using a 648-port Mellanox MLNX-OS CS7500 switch.

Workloads. For the performance evaluation, we focus on evaluating GPT-2 [19] like Transformer based models [30]. We vary the hidden dimension and the number of Transformer blocks to obtain models with a different number of parameters. Note that scaling the depth alone is often not sufficient because it would make training more difficult [12]. Table 3 shows the configuration parameters used in our experiments.

For convergence analysis, such as the delayed parameter update, we use GPT-2 [19] and BERT [6], both of which are commonly used as pre-trained language models and have demonstrated superior performance in many NLP tasks (e.g., natural language understanding and inference) than recurrent neural networks or convolutional neural networks. We use BERT-large, same as the one from [6], which has 24-layer, 1024-hidden, 16-heads, and 336M parameters. Similar

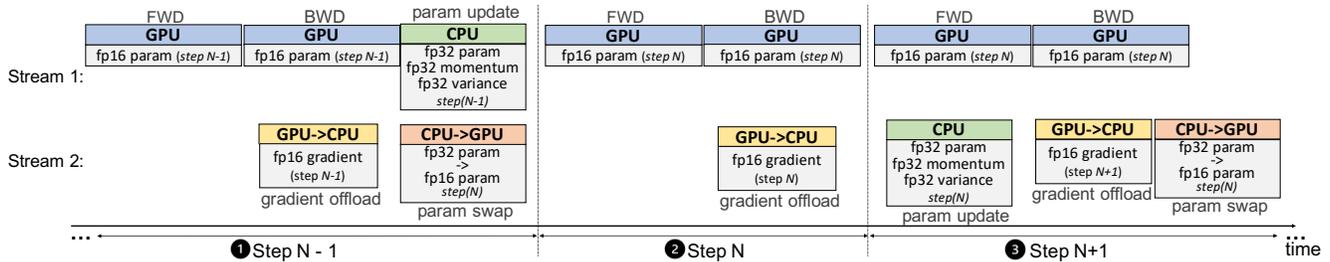


Figure 6: Delayed parameter update during the training process.

Table 3: Model configuration in evaluation.

# params	batch size per GPU	MP setting in ZeRO-Offload	# layer	hidden size
1, 2 billion	32	1	20, 40	2048
4 billion	32	1	64	2304
6, 8 billion	16	1	53, 72	3072
10, 11 billion	10, 8	1	50, 55	4096
12, 13 billion	4	1	60, 65	4096
15 billion	8	2	78	4096
20, 40, 60 billion	8	2	25, 50, 75	8192
70 billion	8	8	69	9216

to [21, 28], we fine-tune BERT on the Stanford Question Answering Dataset (SQuAD) [1], which is one of the most widely used reading comprehension benchmark [22]. Unless otherwise stated, we follow the same training procedure and hyperparameter settings as in [6, 19].

Baseline. We compare the effectiveness of ZeRO-Offload with state-of-arts multi-billion parameter training solutions:

- PyTorch DDP: This is the existing PyTorch Transformer implementation using DistributedDataParallel [14].
- Megatron [28]: One of the current state-of-the-art multi-billion parameter model training solutions, which employs model parallelism to train up to 8.3B parameter models using 512 GPUs.
- SwapAdvisor [9]: SwapAdvisor explores a genetic algorithm to guide model-agnostic tensor swapping between GPU and CPU memory for GPU memory saving.
- L2L [18]: L2L enables training of deep Transformer networks by keeping one Transformer block at a time in GPU memory and only moves tensors in the upcoming Transformer block into GPU memory when needed.
- ZeRO-2 [21]: ZeRO extends data parallelism by eliminating memory redundancies across multiple GPUs, allowing to train models up to 170B parameters with high training throughput using 25 DGX-2 nodes. ZeRO-2 achieves the SOTA results for large model training and is a strong baseline.

6.2 Experimental Results

6.2.1 Model Scale

As an important step toward democratizing large model training, in this part, we first test the largest trainable models on a

single GPU as well as 16 GPUs in a single DGX-2 node.

Single GPU. The largest model can be trained using PyTorch DDP on a single GPU with 32GB memory is 1.4B, before running out of memory, as shown in figure 7. Both Megatron and ZeRO-2 do not increase the trainable model size on a single GPU in comparison to PyTorch, because they both utilize the aggregated GPU memory to fit larger models. In contrast, ZeRO-Offload enables 13B model training on a single GPU, which is more than 9X larger than using PyTorch, Megatron, and ZeRO-2. This is mainly because of ZeRO-Offload’s strategy for maximizing the memory savings on GPU by offloading expensive states such as optimizer states and the majority of gradients to CPU memory. The largest model can be trained with SwapAdvisor on a single GPU is 8B, which is 38% smaller than the model can be trained with ZeRO-Offload. SwapAdvisor relies on a black-box approach and uses a simulator to predict which tensors are more frequently used in order to keep them in GPU memory to maximize training throughput. The prediction can not be fully accurate, and therefore SwapAdvisor keeps more tensors in GPU memory than ZeRO-Offload does. On the other hand, L2L is able to train even larger models (e.g., 17B) on a single GPU by frequently moving weights from unused layers to CPU memory. However, the largest model size does not increase when training L2L with multiple GPUs, which is discussed next.

Multi-GPU in single DGX-2. We further perform model scale tests with 4 and 16 GPUs in a single DGX-2 node, respectively. As shown in Figure 7, the maximum trainable model size stays the same for PyTorch, L2L and SwapAdvisor, because all of them do not handle memory redundancies in data parallelism. As a result, their scalability is bounded by the model scale on a single GPU. Both Megatron and ZeRO-2 support large model training with more GPUs, but they cannot scale efficiently beyond 15B parameters, even with 16 GPUs. Megatron supports larger models than ZeRO-2, because ZeRO-2 still incurs memory redundancies on model weights. On the other hand, ZeRO-Offload easily enables training of up to 70B parameter models by partitioning and offloading optimizer states and gradients to CPU memory

combined with model parallelism. Overall, ZeRO-Offload increases the model scale on a single DGX-2 node by 50X, 4.5X, 7.8X, and 4.2X than using PyTorch, Megatron, ZeRO-2, and L2L, respectively.

6.2.2 Training Throughput

Single GPU. Next, we compare the training throughput of SwapAdvisor, L2L and ZeRO-Offload, for models with billion-scale parameters, on a single GPU. We do not include Megatron and ZeRO-2 in this comparison, because both of them cannot train models bigger than 1.4B parameters due to OOM. We evaluate SwapAdvisor, L2L and ZeRO-Offload with the same training batch size (e.g., 512) and same micro-batch sizes (shown in table 3), with gradient accumulation enabled. We also disable delayed parameter update in this experiment so that the comparison is only from the system efficiency perspective. We evaluate the performance improvement and its impact on the convergence of delayed parameter update in Section 6.2.4.

Figure 8 shows that ZeRO-Offload outperforms SwapAdvisor by 23% (up to 37%) in training throughput. SwapAdvisor relies on genetic algorithm to make tensor swapping decision, which takes hours to find an optimal tensor swapping solution in terms of maximizing the overlapping of computation and tensor swapping. Before getting the optimal tensor swapping solution, SwapAdvisor tries random tensor swapping solutions and hurts training performance.

Figure 8 shows that ZeRO-Offload outperforms L2L by 14% on average (up to 22%) in throughput (TFLOPS). The performance benefit of ZeRO-Offload comes from the following two aspects. First, ZeRO-Offload has a lower communication cost between CPU and GPU than L2L. For a model with M parameters, L2L requires $28M$ data communication volume between GPU and CPU, which is a sum of the weights, gradients, and optimizer states of each layer of the model. As analyzed in Sec. 4.1, the communication volume between CPU and GPU memory in ZeRO-Offload is $4M$, which is 7x smaller than L2L. The reduced communication volume significantly mitigates the bottleneck from CPU-GPU communication. Second, compared with L2L, the parameter update of ZeRO-Offload happens on CPU instead of GPU, but our optimized CPU-Adam implementation achieves a quite comparable parameter update performance than the PyTorch Adam implementation on GPU (evaluated in Sec. 6.2.4). Therefore, although the optimizer update on GPU in L2L is slightly faster than the optimizer update on CPU in ZeRO-Offload, the communication overhead introduced by L2L leads to an overall slower throughput than ZeRO-Offload.

Multi-GPU in single DGX-2. Next, we compare the training throughput of PyTorch, ZeRO-2, Megatron, ZeRO-Offload without model parallelism (w/o MP), and ZeRO-Offload with model parallelism (w/ MP) in one DGX-2 node. When using MP, we use a MP degree that gives the best performance

for both baseline and ZeRO-Offload. We use a total batch size of 512 for all the experiments using a combination of micro-batch per GPU and gradient accumulation. To get the best performance for each configuration, we use the largest micro batch that it can support without OOM. We exclude L2L [29] in this test because its implementation does not support multi-GPU training.

Figure 10 shows the throughput per GPU results when training on multiple GPUs. We make the following observations:

- For 1B to 15B models, ZeRO-Offload achieves the highest throughput and has up to 1.33X, 1.11X, 1.64X higher speeds than PyTorch, ZeRO-2, and Megatron, respectively. By offloading all the optimizer states to CPU with low overhead, ZeRO-Offload can train with larger micro-batch sizes giving higher throughput.
- ZeRO-2 runs out of memory once the model size is beyond 8B due to lack of enough aggregated GPU memory to store the model states on 16 GPUs. Instead, ZeRO-Offload scales to 13B, without model parallelism because it offloads optimizer states and the majority of gradients to CPU memory.
- When combined with model parallelism, ZeRO-Offload enables training up to 70B parameter models with more than 30 TFLOPS throughput per GPU. In contrast, Megatron supports only up to 15B parameter models before running out of memory, using just model parallelism.
- Compared ZeRO-Offload with ZeRO-2 and Megatron, ZeRO-Offload outperforms ZeRO-2 and Megatron in throughput for 1–8B and 1–13B parameter models, respectively. ZeRO-Offload is faster than Megatron, because it eliminates frequent communication between different GPUs and can train with larger micro batch sizes. ZeRO-Offload outperforms ZeRO-2 also due to larger micro batch sizes.

6.2.3 Throughput Scalability

We compare the throughput scalability of ZeRO-2 and ZeRO-Offload³ on up to 128 GPUs in Figure 11 and make the following key observations: First, ZeRO-Offload achieves near perfect linear speedup in terms of aggregated throughput (green line) running at over 30 TFlops per GPU (blue bars). Second, from 1 to 16 GPUs, while ZeRO-2 runs out of memory, ZeRO-Offload can effectively train the model, turning the model training from infeasible to feasible. Third, with 32 GPUs, ZeRO-Offload slightly outperforms ZeRO-2 in throughput. The improvement comes from additional memory savings on GPU from ZeRO-Offload, which allows training the model with larger batch sizes that lead to increased GPU computation efficiency. Fourth, with more GPUs (such as 64

³We do not include comparison against Megatron because it consistently performs worse than ZeRO-Offload, as shown in Figure 10. Given the communication overhead added by model parallelism, scaling out Megatron training can not achieve higher throughput than ZeRO-Offload even with linear scalability.

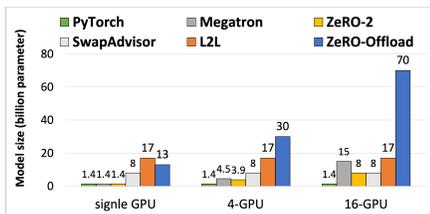


Figure 7: The size of the biggest model that can be trained on single GPU, 4 and 16 GPUs (one DGX-2 node).

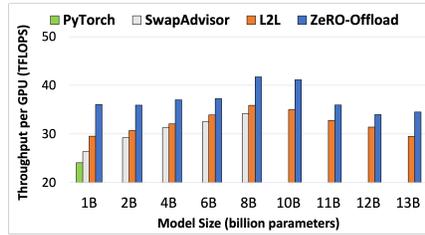


Figure 8: The training throughput with PyTorch, L2L, SwapAdvisor and ZeRO-Offload on a single GPU with a batch size of 512.

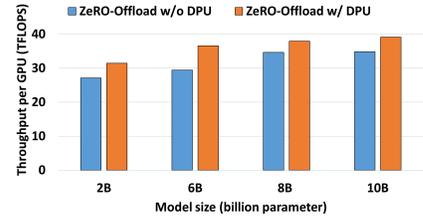


Figure 9: The training throughput is compared for w/o DPU and w/ DPU to GPT-2. Batch size is set to 8.

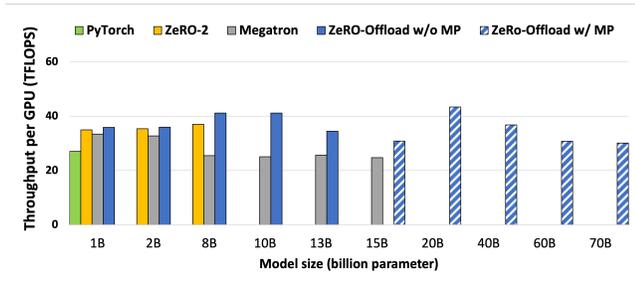


Figure 10: Training throughput with PyTorch, ZeRO-2, MegatronLM, ZeRO-Offload without model parallelism and ZeRO-Offload with model parallelism.

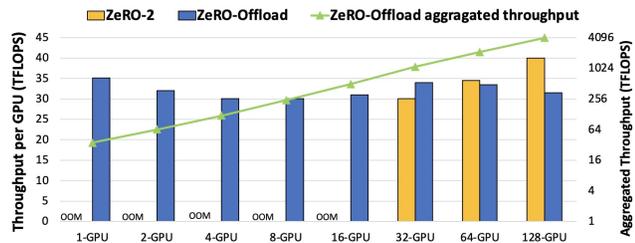


Figure 11: Comparison of training throughput between ZeRO-Offload and ZeRO-2 using 1–128 GPUs for a 10B parameter GPT2.

and 128), ZeRO-2 starts to outperform ZeRO-Offload, because both can now run similar batch sizes, achieving similar computation efficiency, whereas ZeRO-2 does not suffer from the additional overhead of CPU-GPU communication. In summary, ZeRO-Offload complements ZeRO-2 and enables large model training from a single device to thousands of devices with good computation efficiency.

6.2.4 Optimized CPU Execution

A. CPU-Adam efficiency. In this part, we evaluate our Adam implementation against the PyTorch Adam on CPU. Table 4 shows the optimizer execution time of the two implementations for model parameters from 1 to 10 billion. Compared to PyTorch (PT-CPU), CPU-Adam reduces the execution time by over 5X for all the configurations and 6.4X

for the case with 1B parameters. The CPU-Adam optimizer achieves high speedups by exploiting the instruction-level parallelism, thread-level parallelism, and the tile-based data copy scheme (as shown in line 15 of Algorithm 1). Meanwhile, although CPU-Adam has a slower speed than the PyTorch Adam implementation on GPU (PT-GPU), the performance gap is not very huge, and the CPU computation is not a bottleneck of the training throughput.

B. One-step Delayed parameter update (DPU). Figure 9 shows the comparison of the training throughput of GPT-2 with and without DPU. As shown, with DPU enabled, the training achieves 1.12–1.59, updated times higher throughput than without it, for a wide range of model sizes for a small micro batch size of 8. This is expected because DPU allows the optimizer updates to overlap with the next forward computation such that the GPU does not have to be slowed down by the CPU computation and CPU-GPU communication. But, what about accuracy?

Convergence impact We study the convergence impact of DPU on both GPT-2 and BERT. Figure 12 shows the pre-training loss curves over 100K training iterations using PyTorch (unmodified GPT-2), and Figure 13 shows the loss curves of fine-tuning Bert-large model on SQuAD using ZeRO-Offload without DPU, and ZeRO-Offload with DPU. In both cases, DPU is enabled after 40 iterations allowing the training to stabilize in its early stage before introducing DPU.

We observe that the training curves of the unmodified GPT-2 and ZeRO-Offload w/o DPU are exactly overlapped, because ZeRO-Offload w/o DPU performs only system optimizations and does not alter training dynamics. On the other hand, the training curve from ZeRO-Offload with DPU converges slightly slower at the very beginning of the training (e.g., barely can be seen at 2K-5K iterations) and quickly catches up after 5K iterations. For the remaining of the training, the training loss matches the original training until the model converges.

For Bert-Large fine-tuning, we can see that although the training losses are not exactly the same, they converge in the same trend and are largely overlapped. Without changing any hyperparameters, ZeRO-Offload + DPU achieves the same

Table 4: Adam latency (s) for PyTorch (PT) and CPU-Adam.

#Parameter	CPU-Adam	PT-CPU	PT-GPU (L2L)
1 billion	0.22	1.39	0.10
2 billion	0.51	2.75	0.26
4 billion	1.03	5.71	0.64
8 billion	2.41	11.93	0.87
10 billion	2.57	14.76	1.00

final F1 score (92.8) as the baseline. From these results on both GPT-2 pretraining, and Bert-Large fine-tuning, we empirically verify that DPU is an effective technique to improve the training throughput of ZeRO-Offload without hurting model convergence and accuracy. The 1-step staleness introduced by DPU is well tolerated by the iterative training process once the model has passed the initial training phase.

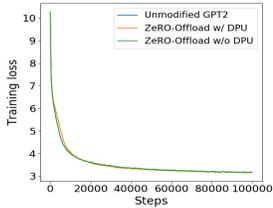


Figure 12: The training loss curve of unmodified GPT-2, ZeRO-Offload w/o DPU and ZeRO-Offload with DPU.

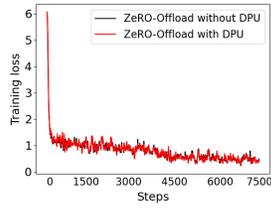


Figure 13: The fine-tuning loss curve of BERT, ZeRO-Offload w/o DPU and ZeRO-Offload with DPU.

6.2.5 Performance Breakdown and Analysis

To better understand the performance benefit from offload strategies and optimization techniques in ZeRO-Offload, we evaluate the training throughput of PyTorch, ZeRO-Offload with PT-CPU, ZeRO-Offload with CPU-Adam (refer as ZeRO-Offload), and ZeRO-Offload with DPU. We perform the evaluation with various batch sizes with 1-billion GPT-2 model on a single GPU. Figure 14 shows the result.

From batch size 1 to 8, PyTorch outperforms ZeRO-Offload with PT-CPU by 16% on average. This is because when the model can fit on GPU memory, PyTorch does not incur any communication overhead. Meanwhile, PyTorch adopts PyTorch GPU Adam (PT-GPU) for optimizer computation on GPU. To reduce the performance loss because of communication and optimizer computation on CPU, ZeRO-Offload optimizes execution on CPU. (1) By optimizing CPU optimizer, ZeRO-Offload implements CPU-Adam and improves the performance by up to 10% compared with using offload strategy only (i.e., ZeRO-Offload with PT-CPU). (2) PyTorch outperforms ZeRO-Offload by 8% on average when the model can fit on GPU memory. As shown in table 4, the performance gap between CPU-Adam and PT-GPU is not very large. Therefore, the performance degradation from PyTorch

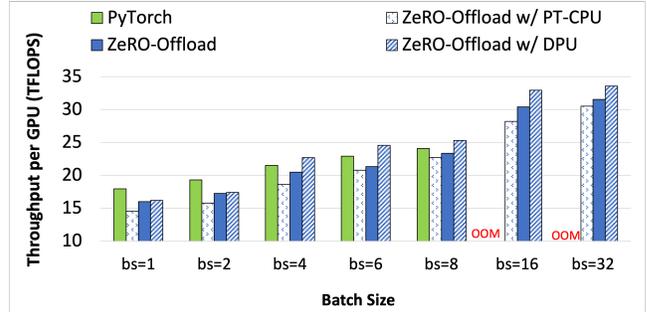


Figure 14: Comparison of training throughput with enabling offload strategies and optimization techniques step-by-step in ZeRO-Offload.

to ZeRO-Offload in Figure 14 are mainly coming from tensor migration overhead between GPU and CPU memory. (3) ZeRO-Offload further introduces one-step delayed parameter update, which overlaps computation on CPU with computation on GPU and improves performance by 7% compared with using ZeRO-Offload without DPU. In summary, leveraging optimized CPU execution, ZeRO-Offload has similar performance as PyTorch when ZeRO-Offload and PyTorch training with the same batch size on GPU.

As the batch size increases, out-of-memory on GPU memory happens in training with PyTorch. The training throughput increases in ZeRO-Offload as the batch size increasing. With unique optimal offload strategy, ZeRO-Offload outperforms PyTorch by 39% for the maximum training throughput that can be achieved on a single GPU with 1-billion model.

7 Conclusions

We presented ZeRO-Offload, a powerful GPU-CPU hybrid DL training technology with high compute efficiency and near linear throughput scalability, that can allow data scientists to train models with multi-billion parameter models even on a single GPU, without requiring any model refactoring. We open-sourced ZeRO-Offload as part of the DeepSpeed library (www.deepspeed.ai) with the hope to democratize large model training, allowing data scientist everywhere to harness the potential of truly massive DL models.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. We thank our shepherd, Mark Silberstein, for his valuable feedback. This work was partially supported by U.S. National Science Foundation (CCF-1718194, CCF-1553645 and OAC-2104116) and the Chameleon Cloud.

References

- [1] The Stanford Question Answering Dataset (SQuAD) leaderboard. <https://rajpurkar.github.io/SQuAD-explorer/>.
- [2] 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, 2020.
- [3] Yu Cao, Wei Bi, Meng Fang, and Dacheng Tao. Pre-trained language models for dialogue generation with multiple input sources, 2020.
- [4] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv: Learning*, 2016.
- [5] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc’auelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012.
- [6] 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.
- [7] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training, 2018.
- [8] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’20*, 2020.
- [9] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’20*, page 1341–1355, New York, NY, USA, 2020. Association for Computing Machinery.
- [10] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, Hyoungho Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2018.
- [11] Hai Jin, Bo Liu, Wenbin Jiang, Yang Ma, Xuanhua Shi, Bingsheng He, and Shaofeng Zhao. Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures. *ACM Trans. Archit. Code Optim.*, 15(3), September 2018.
- [12] 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.
- [13] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.
- [14] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, 2020.
- [15] Gaurav Mitra, Beau Johnston, Alistair Rendell, Eric McCreath, and Jun Zhou. Use of simd vector operations to accelerate application code performance on low-powered arm and intel platforms. pages 1107–1116, 05 2013.
- [16] Nvidia. Automatic Mixed Precision for Deep Learning. <https://developer.nvidia.com/automatic-mixed-precision>, 2019.
- [17] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’20*, page 891–905, New York, NY, USA, 2020. Association for Computing Machinery.
- [18] Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, and Sujeeth Bharadwaj. Training large neural networks with constant memory using a new execution algorithm. June 2020.
- [19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

- [20] 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, 2020.
- [21] 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 (SC)*, 2020.
- [22] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. In Jian Su, Xavier Carreras, and Kevin Duh, editors, *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, pages 2383–2392. The Association for Computational Linguistics, 2016.
- [23] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2020.
- [24] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vdn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-49*, 2016.
- [25] Corby Rosset. Turing-nlg: A 17-billion-parameter language model by microsoft, 2020.
- [26] Christopher J. Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. Measuring the Effects of Data Parallelism on Neural Network Training. *Journal of Machine Learning Research*, 20, 2019.
- [27] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake A. Hechtman. Mesh-tensorflow: Deep learning for supercomputers. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 10435–10444, 2018.
- [28] 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.
- [29] Roman Tezikov. PyTorch implementation of L2L execution algorithm. <https://github.com/TezRomach/layer-to-layer-pytorch>, 2020.
- [30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.
- [31] G. Velkoski, M. Gusev, and S. Ristov. The performance impact analysis of loop unrolling. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 307–312, 2014.
- [32] Oreste Villa, Mark Stephenson, David Nellans, and Stephen Keckler. Buddy Compression: Enabling Larger Memory for Deep Learning and HPC Workloads on GPUs. In *International Symposium on Computer Architecture*, 2020.
- [33] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '18*, page 41–53, New York, NY, USA, 2018. Association for Computing Machinery.
- [34] Junzhe Zhang, Sai-Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. Efficient memory management for gpu-based deep learning systems. *CoRR*, abs/1903.06631, 2019.
- [35] M. A. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized Stochastic Gradient Descent. In *International Conference on Neural Information Processing Systems*, 2010.

Hashing Linearity Enables Relative Path Control in Data Centers

Zhehui Zhang¹, Haiyang Zheng², Jiayao Hu², Xiangning Yu², Chenchen Qi², Xuemei Shi², Guohui Wang²
¹University of California, Los Angeles, ²Alibaba Group

Abstract

A data center network is an environment with rich path diversity, where a large number of paths are available between end-host pairs across multiple tiers of switches. Traffic is split among these paths using ECMP (Equal-Cost Multi-Path routing) for load balancing and failure handling. Although it has been well studied that ECMP has its limitations in traffic polarization and path ambiguity, it remains the most popular multi-path routing mechanism in data centers because it is stateless, simple, and easy to implement in switch ASICs.

In this paper, we analyze the ECMP hash algorithms used in today's data center switch ASICs, aiming for lightweight path control solutions that can address the ECMP limitations without any changes to existing data center routing and transport protocols. Contrary to common perceptions about the randomness of ECMP hashing, we reveal the linear property in the hash algorithms (e.g. XOR and CRC) used in widely deployed switch ASICs in data centers. Based on the hashing linearity, we propose relative path control (RePaC), a new lightweight, and easy-to-deploy path control mechanism that can perform on-demand flow migration with deterministic path offsets. We use a few case studies to show that RePaC can be used to achieve orders of magnitude faster failover and better path planning with up to 3 times link utilization gain in hyper-scale data centers.

1 Introduction

To support the high bandwidth and high availability requirements of cloud and big data applications, data center networks are often designed with rich path diversity. Typical data center network topologies, such as FatTree [1], Clos [10], or Hyper-Cube [16], offer hundreds of paths across multiple tiers of switches between any pair of servers. In such a multi-path network, managing a large number of paths among all end-point pairs is challenging. The path selection mechanism is always a key part of data center network design to fully utilize the high bandwidth from these paths and achieve good traffic load balance and fault-tolerance properties.

The multi-path IP routing protocol (e.g. BGP [27]) with ECMP [20] is the most common routing scheme in many production data centers [13, 26]. In such networks, reachable network prefixes and available paths are propagated to all the switches using BGP, and each switch uses ECMP to select a next hop based on the hash value from specified packet header fields. To understand the performance of ECMP-based routing, much research has been done. A common perception is that ECMP offers reasonable load balancing and fault tolerance among a large number of uniform flows, however, it is difficult to perform any explicit path control on ECMP because it is stateless, and its hashing calculation is random [18, 21, 46]. Besides, ECMP suffers from traffic polarization because it performs static load balancing based on header fields without considering flow sizes [2].

Many alternative path control mechanisms, such as Hedera [2], XPath [21] and Multi-path TCP [36, 37], have proposed to redesign routing and transport layers to better leverage the path diversity. However, these proposals often require a redesign in either the server network stack or routing protocol of data center switches. As a result, although these proposals offer many advantages over ECMP on dynamic load balancing and precise path control, they have seen very limited deployment in today's production data centers. For the path selection mechanism, ECMP remains the common practice because it is stateless, simple, and easy to implement in switch ASICs.

In this work, we take a different approach to address the ECMP limitations in load balancing and failure handling. Given the wide deployment of BGP with ECMP routing, we look for solutions that require minimal changes to server software stack and data center routing protocols. Instead of treating ECMP as a random path selection black-box, we analyze the ECMP path selection process and investigate typical hash algorithms used in the most popular data center switch ASICs on the market. Our study reveals that most widely-deployed switches use XOR, CRC or their variants for ECMP hashing. These hash algorithms hold a linear property ($ECMP(a) \oplus ECMP(b) = ECMP(a \oplus b) \oplus ECMP(0)$),

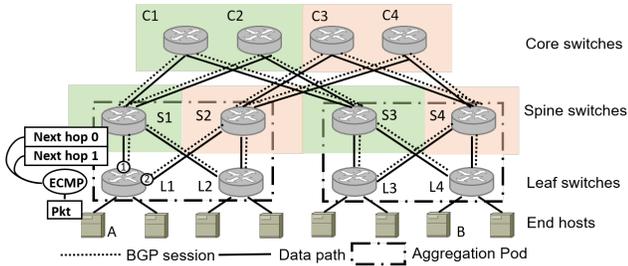


Figure 1: A Clos-based data center network topology

which guarantees a deterministic mapping between packet header changes and path changes. We analyze how hashing linearity affects traffic load balance and how the hashing linearity can be used to control the flow paths to better leverage the path diversity.

We propose a relative path control (RePaC) scheme, which uses the deterministic mapping between the header change and path change to perform on-demand flow migration for failure handling and load balancing. We validate the hashing linearity and relative path control scheme on a testbed with the same ECMP configuration as our production data centers. Our evaluation shows that RePaC can achieve fast failover and better traffic engineering for TCP and RDMA applications. With RePaC, failover speeds up by orders of magnitude compared with existing in-network failover approaches. RePaC also outperforms MPTCP with 2(4) subflows in recovery success rate by 36%(21%). In addition, RePaC-based traffic engineering reduces flow completion time by up to 25% and improves link utilization by up to three times. Proven to be simple and effective, RePaC can be readily deployed in production data centers with no changes required in the routing and transport protocols.

As far as we know, our work is the first study that investigates the linear property of ECMP hash algorithms and discusses its applications to data center path control. We show the feasibility of leveraging the hash algorithm properties to develop flexible path control algorithms for load balancing and failure handling, which contradicts the common perception about the randomness and ambiguity of ECMP path selection. The resulting solution RePaC is lightweight, easy to deploy in production data centers, and easy to integrate with a large spectrum of applications. Our work sheds light on a new perspective of multi-path routing, traffic engineering and application design in data centers.

2 Background and Motivation

2.1 Data Center Networking Primer

Most data center networks adopt some variant of a multi-rooted tree topology. Figure 1 shows an example of a Clos-structured data center network with three tiers of switches, which provides abundant paths between any pair of servers to

achieve high aggregated bandwidth and tolerate potential link and device failures. To better utilize the available paths in the network, multi-path routing is the key part of the data center network design, with load balancing and fault tolerance as two primary design goals.

Multi-path BGP with ECMP is the most common multi-path routing design in hyperscale data centers [27]. In this design, switches establish BGP sessions among each other over the connected links and choose next hops based on BGP routing information. Multiple equal-cost next hops are grouped as ECMP groups when routes are installed into the routing table in the switch ASIC. When a packet reaches a switch, a random next hop will be selected in the ECMP group, so flows can be evenly distributed among parallel links for load balancing. As shown in Figure 1, if a packet is being sent to switch L3 via switch L1, the BGP session running on L1 will first decide all equal-cost next hops, S1 and S2. In ECMP, each switch distributes packets based on a hashing value calculated from specific packet header fields. ECMP would select one from S1 and S2. If ECMP distributes flows to S1 and S2 evenly, load balancing is fulfilled. To support failover, the in-network BGP sessions send keepalive messages to detect whether the link is still available. In case of failure, BGP tears down the session of the malfunctioning link, and the corresponding next hop is removed from ECMP. Therefore ECMP is critical for both load balancing and failure handling in data center networks.

2.2 The Limitations of ECMP

Many previous studies have shown that ECMP has several key limitations in practice. First, ECMP fails to leverage the path diversity in a lot of scenarios. Hedera [2] shows ECMP may cause significant bandwidth loss when flows are not evenly distributed. The situation becomes worse when large and long-lived flows co-exist. Second, ECMP distributes traffic using random hash algorithms, which makes it difficult to perform precise path control. A lot of fine-grained flow scheduling and traffic engineering mechanisms cannot be done in data center networks over ECMP [5, 21].

Due to the aforementioned limitations, a common perception of ECMP is that ECMP provides decent load balancing among a large number of flows evenly distributed across a large header space. However, ECMP works under strong assumptions about traffic patterns, and it may fail in many cases with traffic polarization, slow failover, and under-utilized links. And it is difficult to perform deterministic path control because ECMP path selection is random.

Several previous studies have proposed alternative solutions to address the limitations of ECMP from both routing and transport layers. For example, Hedera [2] is proposed to perform dynamic flow scheduling based on a traffic matrix using OpenFlow. XPath [21] proposes to enable explicit path control for data center applications using pre-installed routes. FUSO [8] proposes a multi-path loss recovery mechanism

on MPTCP to enable fast failure recovery in data centers. However, these solutions require major redesigns of the data center routing protocol or server network stacks. And major redesigns of routing and transport protocols have huge impacts on the data center applications and daily network operations. As a result, these solutions still have very limited deployment in today’s production data centers.

ECMP still has wide deployment in many production data centers for multi-path routing, however, the implementation of the ECMP hashing mechanism remains a mystery for the research community. There isn’t a thorough analysis of the details of the ECMP mechanism on popular data center switch ASICs. Volur [47] is the only study we have seen, which tries to model ECMP as a deterministic mapping from headers to paths. It aggregates all the mappings from all switches and uses model replay to predict ECMP path selection. However, Volur still treats ECMP as a black-box model, and a full network model replay incurs too much overhead for real-time traffic engineering and path control in large data centers.

This dilemma of path control in data centers motivates us to dig into the black-box of ECMP implementation of widely-deployed data center switches and look for lightweight solutions to address the ECMP limitations. Such solutions can be incrementally deployed to a large volume of existing data centers and benefit data center applications without a major redesign of the network. We explore the factors that affect ECMP path selection and reveal an interesting property, linearity of hash algorithms used by most merchandise switch ASICs, which could be used to enable a new relative path control scheme.

3 Demystifying ECMP Path Selection

Our analysis of ECMP path selection is based on the ECMP implementation of popular switch ASICs available on the data center switch market. We look into the publicly available open source repositories and documents about the ECMP implementation on widely deployed switch ASICs [6, 12, 31, 33]. We also investigate the ECMP configurations from the widely-adopted open switch abstraction interface (SAI) [34], which is supported by most switch ASIC vendors, such as Broadcom, Barefoot, CISCO, Mellanox, and Marvell [29]. We consider the variations of ECMP implementations [20, 32, 38] and validate our analysis with commercial switches. Specifically, our validation covers Broadcom Tomahawk series, Trident series, and Barefoot Tofino series [7].

3.1 Modeling ECMP Path Selection

ECMP is typically implemented as part of the routing table matching stage in the switch ASIC pipeline. When multiple routes are available for a given network prefix, equal-cost next hops are added into an ECMP group in the routing table. When a routing table entry is matched for an incoming

packet, the packet will be forwarded to one of the next hops in its ECMP group, based on the hashing of the packet header fields [33].

Figure 2 shows a general ECMP processing model from typical switch ASICs. The input of ECMP is header fields, and the output is a next hop ID. There are four stages in ECMP processing: pre-processing, hashing, post-processing, and bucket mapping. We use the following functions to denote the ECMP stages: $Pre_proc()$, $Hash()$, $Post_proc()$, $Bucket_B^N()$. The overall ECMP processing can be described as:

$$ECMP(h) = Bucket_B^N(Post_proc(Hash(Pre_proc(h)))) \quad (1)$$

where the hash function plays a key role in path selection.

In the $Pre_proc(h)$ stage, the H -bit input packet header fields h , with range $\{0, 1\}^H$, together with a configurable hash seed are processed using bit-wise operations, such as AND, XOR, shifting and masking. For example, if we use source and destination IP addresses and port numbers as input, H would be 96. Then $Hash()$ denotes the hash function from $\{0, 1\}^I$ to $\{0, 1\}^O$, where I is the length of the pre-processed input and O is the length of the output hash result, both in number of bits. O is usually 32 or 16 since most switches use 32-bit or 16-bit values to represent hash results [12, 31]. In the $Post_proc()$ stage, the hash result is further shuffled using bit-wise operations. Then $Bucket_B^N()$ performs modulo operations to map the post-processed hash result $\{0, 1\}^O$ to one of the next hops in the ECMP group $\{0, \dots, N - 1\}$ placed in B hash buckets, where N is the number of next hops. The entire ECMP process is a mapping from $\{0, 1\}^H$ to a number in $\{0, \dots, N - 1\}$.

3.2 The Linearity of ECMP Process

In this section, we discuss the linear property of the most common ECMP hash functions, and how linearity impacts the path selection.

Hash functions overview: The commercial switch ASICs support various hash functions for ECMP, such as random, XOR, CRC and Pearson hashing [11, 24, 31]. Among them, CRC and XOR are two popular hash algorithms defined in SAI [34], and supported by most switch vendors [29]. There are two reasons that CRC and XOR are widely adopted. These two algorithms have been used in communication systems with mature and efficient ASIC implementation, which includes plenty of circuit optimization [31, 42, 45]. In addition, previous analysis has shown that CRC and XOR algorithms have good load balancing performance given a uniform distribution of flows [49].

The SAI [34] also cites the random hash algorithm, but it is not commonly used in the ECMP implementation of modern switches. The reasons are twofold. First, packet-level random path selection will result in a nightmare in network monitoring

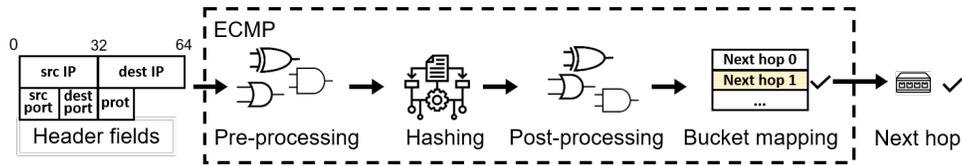


Figure 2: An example of an ECMP packet processing pipeline: pre-process certain header fields, hash and post-process the hash results to get bucket number two, then refer to the bucket for the next hop ID

and troubleshooting. Second, packets from the same flow need to be hashed onto the same path to avoid the out-of-order issues that affect end-to-end transport performance. There are other hash algorithms that are not included in SAI, such as Pearson hashing, which are proprietary of a specific switch ASIC vendor. In this paper, we will focus on the properties of the most common hash functions using CRC and XOR.

The linearity of hash functions: We define the hashing linearity as follows: a hash function is linear, if

$$\text{Hash}(h_i) \oplus \text{Hash}(h_j) = \text{Hash}(h_i \oplus h_j) \oplus \text{Hash}(0) \quad (2)$$

where h_i and h_j are arbitrary packet headers, and $\text{Hash}(0)$ is the hash result for a packet with all 0's, which is a constant given the initial hashing seed. Both XOR and CRC hashing satisfy hashing linearity. The detailed proof of CRC/XOR hash linearity can be found in supplemental materials [48].

Insight: If the linear hash function is fixed, for any packet header h_i , the hash value after a relative change Δ on the original header fields is deterministic, i.e. $\text{Hash}(h_i \oplus \Delta) = \text{Hash}(h_i) \oplus \text{Hash}(\Delta) \oplus \text{Hash}(0)$. In other words, as long as we know the mapping from Δ to $\text{Hash}(\Delta)$ and $\text{Hash}(0)$, we can predict the relative hash value change.

The linearity of $\text{ECMP}()$: We prove the linearity of ECMP by proving the linearity of the other three procedures $\text{Pre_proc}()$, $\text{Post_proc}()$, and $\text{Bucket}_B^N()$. The pre-processing and post-processing functions are bit-wise operations used to shuffle certain fields of the packet header or hash result to cope with hash polarization [20, 24, 31]. Hash polarization would cause load imbalance when hashing results favor certain buckets over others. Pre-processing usually uses bit-wise operations such as AND, XOR, bit shifting and masking. Post-processing uses XOR-folding of 32-bit hash results to get a 16-bit result [38]. Our analysis shows that these bit-wise operations in pre-processing and post-processing functions do not affect the linear property of ECMP path selection. The detailed proof can be found in supplemental materials [48].

Linearity also holds for the $\text{Bucket}_B^N()$ function if N and B are both powers of two, which is expected if a fat-tree topology is used. Then the modulo operation is equivalent to a bit-wise shifting, which is proved to be linear. There are two scenarios in which linearity does not apply for ECMP. The first scenario is that the next hop ID of the Bucket is randomized. However, we can reorder buckets based on the next hop ID after bucket mapping updates, which is commonly adopted for consistent hashing. The second scenario is that the number of buckets is not a power of two 2^k , where k is an integer. The number of buckets depends on the topology.

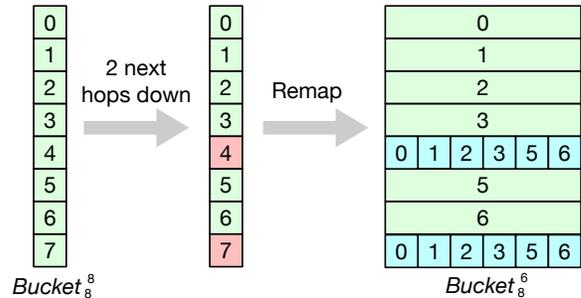


Figure 3: Consistent bucket mapping upon 2 link failures. Buckets of failed hops are remapped to 6 virtual entries.

Table 1: Probability of different failure scenarios with various # of failed links and probability that linearity holds (\mathbb{P}) under various link failure ratios σ ($B = 8$)

# failed links	0	1	2	3	\mathbb{P}
$\sigma=1\%$	92.2%	7.45%	0.26%	0.005%	99.68%
$\sigma=0.1\%$	99.2%	0.79%	0.003%	$< 10^{-9}$	99.90%
$\sigma=0.01\%$	99.9%	0.08%	$< 10^{-6}$	$< 10^{-10}$	99.99%

If a fat-tree topology is used, the number of buckets is 2^k . If the data center adopts a variation of a fat-tree topology, each switch can enable consistent hashing by adding virtual nodes to make the bucket number as 2^k [33].

Thus we have

$$\text{ECMP}(h_j \oplus \Delta) = \text{ECMP}(h_i) \oplus \text{ECMP}(\Delta) \oplus \text{ECMP}(0) \quad (3)$$

where h_i and h_j are arbitrary packet headers, $\text{ECMP}(0)$ is the output given an all 0's header, a constant for a given ECMP configuration.

Linearity upon link and device failures: In the case of link and device failures, the number of next hops N varies and may not always be a power of two, which breaks the linearity of the whole ECMP process. We leverage consistent hashing and bucket remapping to guarantee linearity for most links in the ECMP group. We illustrate our ideas with an example with two link failures. In the case of failures, for example both next hop 4 and 7 fail as shown in Figure 3, removing two buckets (Bucket_6^8) will break linearity since 6 is not a power of two. If we keep these two buckets and simply adopt consistent hashing to remap these two buckets to two random available next hops, the load is imbalanced given 8 buckets are mapped to 6 hops. We thus remap these two buckets to 6 virtual entries and map these entries to available next hops uniformly as shown in Figure 3. Therefore, we can still

preserve the linearity for available links by fixing the hash bucket B as a power of two. The load balancing performance is equivalent to an ECMP table with size of the least common multiples of B and N . And the $Bucket_B^N()$ function can be modeled as follows:

$$Bucket_B^N() = Bucket_B^B() \text{ is down? } Bucket_N^N() : Bucket_B^B() \quad (4)$$

where $Bucket_B^B()$ holds linearity as B is a power of two. And the non-linear function $Bucket_N^N()$ is a modulo operation over N followed by a mapping from $\{0, \dots, N-1\}$ to all available next hop IDs.

As stated in [15], links and devices inside data center networks usually have four 9's of high availability. Before BGP updates bucket mapping upon the first link failure, linearity still holds since the output of $Bucket()$ remains the same. The linearity-based path control might fail upon more failures, but the probability is low. For example, the probability that more than 1 failure happens in the same ECMP group is only 41% [15]. We analyzed the probability that linearity holds under various link failure ratios as shown in Table 1. We consider the probability of occurrence with various numbers of failed links under different link failure ratios. Given F next hops removed due to failure links, $Bucket_B^N()$ holds linearity with a probability of $1 - F/B$. Based on our observation of 1%-0.01% failure rate σ in our production data centers of more than 1500 links, the probability that linearity holds is more than 99% even when link failures are considered. Here we assume link failures are independent. If we assume the probability of link failures under the same ECMP group is positively correlated, the probability of recovery decreases. However, according to previous studies [15] and our observations in our own data centers, the probability of concurrent failures in one ECMP group is low.

Validation: We validate linearity in commercial switches by checking whether Equation 3 holds. We configure the switch with 8 next hops. First, we generate 1 million packet headers with random IP addresses, port numbers, protocol field and reserved fields. For each header, we record the output of ECMP before and after adding the offset Δ , which are denoted as $ECMP(h_i)$ and $ECMP(h_i \oplus \Delta)$, where h_i are random packet headers and Δ ranges from 0 to $2^k - 1$ for a k bit field. We do the same for another random packet header h_j . Finally, as shown in Figure 4, by comparing whether $ECMP(h_i) \oplus ECMP(h_i \oplus \Delta) = ECMP(h_j) \oplus ECMP(h_j \oplus \Delta), \forall i, j$ holds, we verify that switches from Broadcom and Barefoot all satisfy the linearity. We also compare with different hashing seed configurations. The results show that linearity holds for different seed settings from 0 to $2^{32} - 1$. Linearity holds for validated vendors under XOR hashing, CRC hashing, and variants of XOR/CRC hashing (e.g. optimized hashing by concatenating COR hashing results and CRC hashing results). Besides the Broadcom and Barefoot switches that we validated in the paper, switches from 100+ platforms with SAI also support linear hashing [29, 34].

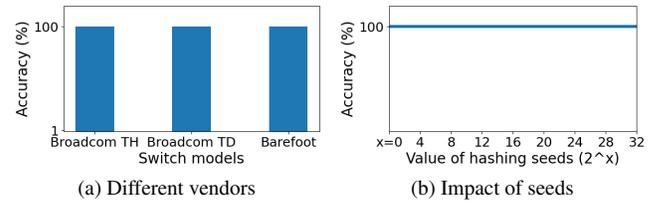


Figure 4: Validation with commercial switches. Accuracy of ECMP linearity-based prediction is 100% in different vendors and under selection of different seeds.

3.3 The Path Linearity

Our previous analysis has demonstrated the linearity of ECMP path selection at a single switch. In this section, we analyze the linearity behavior of ECMP in the multi-hop environment and discuss how the linearity can be used for path control.

Multihop linearity: In the multihop environment, a path is defined as a list of end hosts and switches, for example, $(A \rightarrow L1 \rightarrow S2 \rightarrow C3 \rightarrow S4 \rightarrow L4 \rightarrow B)$ is a path from the end host A to the end host B in Figure 1. In general, a path is modeled as $(sender, s_1, \dots, s_X, receiver), s_i \in \{0, \dots, N_i - 1\}$, where X is the number of hops, N_i is the number of equal-cost next hops at hop i , and s_i is the next hop ID chosen by hop $i - 1$. There can be one or more links between the end host and leaf switches.¹

We compile the path as the concatenation of the binary representation of next hop selections $Path(h) = s_1 \cdot s_2 \cdot \dots \cdot s_X$, where the sender and receiver are omitted. In a typical three-tier data center network, X is 5 for inter-pod traffic and 3 for intra-pod traffic, where a pod is an aggregation of leaf and spine switches as shown in Figure 1. Since ECMP decides the next hop at each switch, we have $s_i = ECMP_{s_{i-1}}(h), i = 1, \dots, X$. In order to calculate a path, we need the ECMP functions of all switches along the path. However, since we are only interested in the linearity, we aim to calculate $Path(h \oplus \Delta)$. As our analysis shows, the linearity $ECMP(h)$ holds at a single hop. For two hops $s_1 \cdot s_2$, concatenation of $s_1 = ECMP_{sender}(h)$ and $s_2 = ECMP_{s_1}(h)$ still satisfies $ECMP_{sender}(h \oplus \Delta) \cdot ECMP_{s_1}(h \oplus \Delta) = (ECMP_{sender}(h) \cdot ECMP_{s_1}(h)) \oplus (ECMP_{sender}(\Delta) \cdot ECMP_{s_1}(\Delta)) \oplus (ECMP_{sender}(0) \cdot ECMP_{s_1}(0))$ under the assumption that the selection of s_1 will not affect $ECMP_{s_1}(\Delta)$ and $ECMP_{s_1}(0)$. Note this assumption holds as long as switches in the same ECMP group are configured with the same hashing configuration. By the same methodology, we can prove the linearity for the concatenation of the entire path. Then we have

$$Path(h \oplus \Delta) = Path(h) \oplus Path(\Delta) \oplus Path(0) \quad (5)$$

¹ If there are several leaf switches to choose, the end host usually adopts Link Aggregation Control Protocol (LACP) to choose the next hop based on the hashing result of packet header fields like ECMP [4, 9]. Our analysis on ECMP linearity applies to LACP.

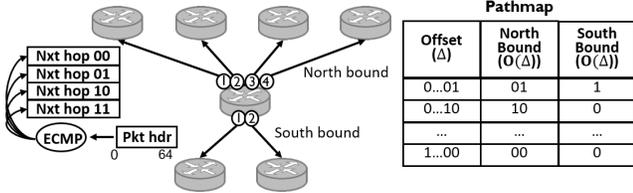


Figure 5: Illustration of pathmap. There are 64 offsets for a header space with 64 bits. Each row represents the path change $O(\Delta)$ for northbound and southbound with offset Δ .

Pathmap: Equation 5 suggests that with the linearity of ECMP path, given a relative change to an arbitrary packet header ($h \oplus \Delta$), we can predict the corresponding ECMP path change as $Path(h \oplus \Delta) = Path(h) \oplus O(\Delta)$, where $O(\Delta) = Path(\Delta) \oplus Path(0)$. This mapping relationship from Δ to $O(\Delta)$ can be modeled with a pathmap structure. As shown in Figure 5, we use a $K \times 2$ table to represent the pathmap of a single switch, where K is the number of bits used for path control, and each column corresponds with the northbound and southbound path offset $O(\Delta)$ for corresponding header offset Δ . If there are N hops, we need a $K \times N$ table. Here we use K entries since we can represent the entire header space by XORing Δ , where Δ are headers $0...010...0$ with only one significant bit. With this model, we compress the ECMP model exponentially from the header space 2^K . In most well-defined packet headers, the number of bits that can be used for path control is limited, which restricts the size of the pathmap.

Insights: Given the proven linearity, we can control the relative path change by changing certain header fields. This is feasible since hashing linearity is a built-in property for linear hashing algorithms. To extend hashing linearity to path linearity and improve accuracy upon link failure, we also need to adopt several ECMP hashing configurations, i.e. consistent hashing, bucket remapping and the same hashing configuration for the same ECMP group².

4 Relative Path Control

4.1 Design

Our analysis of the ECMP hash linearity suggests a deterministic mapping between the packet header changes and path changes. It provides a powerful tool to predict the relative path offset $O(\Delta)$ based on a packet header offset Δ .

We propose a relative path control algorithm, RePaC, based on the insight from hashing linearity. RePaC has two parts: offline pathmap collection and online path control. The offline pathmap collection module acquires the mapping from relative header change to the relative path change based on the static configuration of the network. The online module

²Supplemental materials [48] discuss a solution with relaxed requirements that allow different hashing seeds.

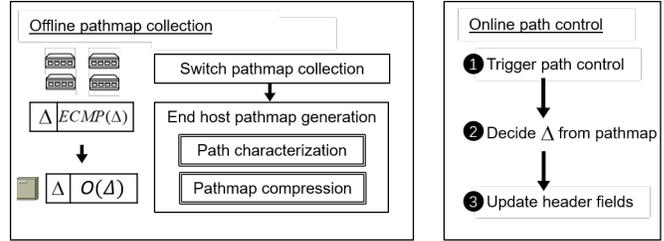


Figure 6: Relative Path Control Process

decides the header change accordingly and alters packets to navigate certain flows to a different path with deterministic offset. The overall procedure is shown in Figure 6.

Offline pathmap collection: In order to collect the pathmap for a switch as shown in Figure 5, we need to send the packets with reference headers with different offsets Δ to northbound and southbound egress ports on the switch. As introduced in Section 3.3, Δ is $0...010...0$ with a single significant bit, and the position of the significant bit ranges from 1 to K , where K is the number of controllable bits in the packet header. The pathmap of a switch depends on the static ECMP hash configuration. So to collect the pathmap for a whole data center network, we need to consider different hashing algorithms (e.g. XOR, CRC with different polynomials) and different hashing seeds. In production data centers, the switches in the same tiers of the network are often designed to use the same ECMP configurations for the simplicity of network management and operations. Then we only need to collect pathmaps for three types of ECMP hash configurations for a typical data center network in a 3-tier Clos topology. Note if the hashing algorithm is fixed, the mapping between the hashing seed and pathmap is deterministic, which reduces the overhead of collecting pathmaps from switches with different hashing seeds. The offline pathmap collection can be easily done on today's network management systems.

Online path control: The online path control module has two parts, the triggering function and the decision function. The triggering function is triggered when path control is needed, for example, upon failures. The decision function decides the path header offset. Users can design the decision function based on application requirements. We define a utility function $util(\Delta)$ for path control. The target of path control is $\hat{\Delta} = argmax(util(\Delta))$. For example, if the target is to change the path, we define $util(\Delta) = Path_changed(\Delta)$, where $Path_changed(\Delta)$ is a function to check whether applying header offset Δ could change the path. We showcase how to design the utility function in Section 4.2.

Based on the designed utility function, the pathmap can be reconstructed in different formats to facilitate searching for an offset for a specified relative path change. For example, a pathmap can be compressed into a hashmap with keys as desired path changes and values as all possible header offsets. For solutions that are not sensitive to the value of relative path change, for example, a failover solution that aims to use

a specific path change, the pathmap can be compressed to a list of desired header offsets that can lead to the desired path change. The structure of the pathmap is decided by the granularity of path control required by applications.

Another key question is which header fields to use for path control. Options depend on what user-defined fields are configured for ECMP hashing. By default, switches might only configure IP addresses and port numbers for ECMP hashing. Based on [28], SAI-supported switches can configure all header fields for hashing. Among all the fields, the high significant bits of TTL fields (8 bits) are typically usable since the number of hops in data centers is small. If switches do not support user-defined fields, the source port number (16 bits) can be leveraged though it might require changes of the port allocation. Besides, the source IP address can be controlled with certain flexibility in incast scenarios if there is a configurable DHCP server inside the pod to assign IP addresses dynamically.

4.2 The Applications of Relative Path Control

In this section, we present two case studies to demonstrate the applications of relative path control.

4.2.1 RePaC for fast failover

Leveraging relative path control, we design a lightweight and fast failover mechanism to detect path failures at the transport layer and update the packet headers to migrate the traffic from the failed path. Our approach relates to the previous end-host-based failure recovery [19, 23] but gives strong guarantees on path control with the ECMP hashing linearity and does not require any changes to the network protocols and switch hardware. We demonstrate a failover mechanism with TCP traffic as an example. The RePaC-based failover mechanism generally applies to any network transport layer protocol with a retransmission mechanism.

The RePaC-based fast failover involves retransmission-triggered failure detection and failure recovery by altering packet headers. It works as shown in Algorithm 1. Our algorithm detects the second retransmission of each flow to trigger failure recovery, then every packet in that flow will be marked for path control. We choose the higher four bits of TTL fields and TCP five tuples as input for $ECMP()$. In this case, we can vary the higher four bits of TTL fields, so Δ is in $\{0001, 0010, 0100, 1000\}$.

After failure recovery is triggered, we mark the reserved bits with a different value to reroute this packet to a different path. In theory, if the path offset $O(\Delta)$ is non-zero at each hop, we can ensure the new path $Path(h \oplus \Delta)$ has no overlapping links or devices with $Path(h)$, where h is the packet header of the flow affected by the failure. With RePaC, we can generate a path mapping table from all marking values to the

Algorithm 1 Failure recovery

Input: Packet p and corresponding flow f
Output: Packet p with $f.mark$ within path marking period.
 Otherwise, original packet p

- 1: **if** p is a repeated retransmission **then**
- 2: Update $f.mark = \text{SelectNextMark}(p)$
- 3: Begin path marking period
- 4: **end if**

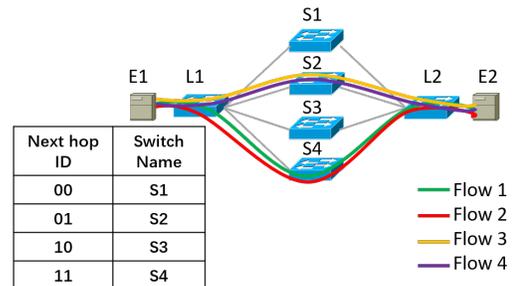


Figure 7: Example network. Flow polarization causes under-utilization of paths.

network path change, as illustrated in Figure 5. And we design the utility function as $util(\Delta) = Path_changed(\Delta)$, where $Path_changed(\Delta)$ is a simple function to check whether $O(\Delta)$ is non-zero for each hop, which indicates that the next hop selected at each hop would be different from the previous path. Then we use a subset of eligible Δ as marking values for fast failover purposes. If we cannot find an $O(\Delta)$ with all hops changed, we select the one with the most changes. RePaC will stop path marking after a configurable timer. We set the timer as the failure detection time in BGP. Path marking will be triggered again if BGP fails to recover the failure.

4.2.2 RePaC for traffic engineering

Given the multi-path nature of data center networks, traffic engineering, the ability to schedule traffic on many non-overlapped paths to better leverage the network bandwidth, has been a key part of data center network design [2, 5, 30, 37]. Compared to existing proposals, such as Hedera [2], MicroTE [5], and XPath [21], RePaC provides a lightweight tool for us to plan and distribute flows in data center networks for traffic engineering without any changes to the switch hardware and routing protocols.

Path planning using RePaC: Given the topology and ECMP configurations of the network, we derive the pathmap M , a map between each path offset $O(\Delta)$ and the corresponding header offset Δ , by sending probing packets with different Δ . Here we show an example for the network in Figure 7. We consider the lower 6 bits of src_port is controllable and probe every switch L1 for $O(\Delta)$ to get Table 2. Table 2 shows how changing a bit in src_port changes the selected hop ID. XORing all possible hop ID changes results in four different path offsets.

Table 2: Probed results. Six controllable bits generate six headers Δ , each with one significant bit. Then RePaC uses six headers to get $O(\Delta)$.

Δ	000001	000010	000100	001000	010000	100000
$O(\Delta)$	00	00	01	00	01	10

Table 3: Calculated $M = (O(\Delta), \Delta)$ for path planning. For each $\Delta = 0, \dots, 63$, RePaC calculates the Path ID.

$O(\Delta)$	0	1	2	3
Δ	0-3, 8-11, 20-23, 28-31	4-7, 12-19, 24-27	32-35,40- 43, 52-55, 60-63	36-39, 44-51, 56-59

We then derive all the possible header changes for each path offset $O(\Delta)$. For example, $\Delta = 000100 \oplus 100000$ is mapped to path offset $01 \oplus 10 = 11$. We index these path offsets as Path IDs to differentiate paths for path planning. Finally, We get the pathmap M as shown in Table 3, which is a map from Path ID $O(\Delta)$ to a set of Δ sharing the same path offset $O(\Delta)$.

From Table 3, we can observe that the selected Path IDs are limited to two choices for src_port values from 0 to 31. This means if E1 sends flows with src_port ranges from 0 to 31, ECMP will select only half of the next hops. We conjecture that only a subset of bits in the hash results are selected by the post-processing function for final ECMP resolution. This suggests that the default ECMP path selection could result in only half of the paths being utilized in the example scenario. Based on the pathmap M in Table 3, we can assign src_port values (e.g. $i, i \oplus 4, i \oplus 32, i \oplus 36$) for those 4 flows to ensure that all four possible paths are utilized.

Based on the analysis in Section 3.3, we can infer the path offset between two flows by looking up the Δ of their packet headers in the pathmap. Therefore, we can use a valid packet header h_{ref} as a reference, then for each $flow_i$, look up the desired path offset in the pathmap for unused Δ and assign $h_{ref} \oplus \Delta$ as packet header h_i . We plan the paths for all flows based on estimated loads as shown in Algorithm 2. In the example network of Figure 7, flows might conflict with each other according to default ECMP. With our algorithm, flows are evenly distributed on diverse paths. Note we can extend the algorithm to consider both the request load and the response load. The pathmap at the receiver can predict the path for response load.

5 Evaluation

5.1 Implementation and Test-Bed Setup

We implement RePaC as a library on the servers to perform on-demand path control. There are three components in the implementation, a pathmap collector that generates a pathmap database given a network, a module that monitors all outgoing

Algorithm 2 Load based path planning

Input: Pathmap M as illustrated in Table 3,

Output: assign a packet header h_i for $flow_i$ to minimize the deviation of bandwidth utilization L_p for each path

- 1: $h_{ref} \leftarrow$ a random valid packet header; $L_p \leftarrow 0$ for all paths
- 2: **for** $flow_i$; in all flows **do**
- 3: Find $\Delta \in M$ for Path ID p with the minimum load L_p
- 4: $h_i = h_{ref} \oplus \Delta$
- 5: $L_p +=$ estimated bandwidth utilization for $flow_i$
- 6: **end for**

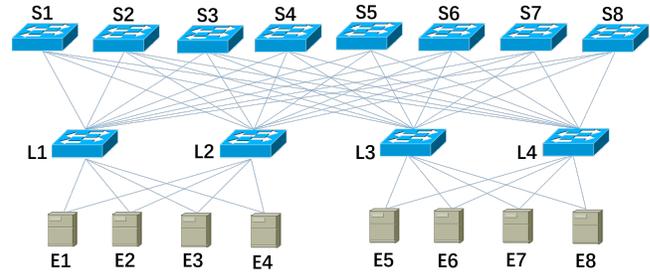


Figure 8: Test topology

traffic on servers to perform TCP failover, and a path planner application that generates the best paths for applications. We implement flow monitoring with the VNIC, which maintains a soft stateful flow table. For each TCP flow, RePaC checks for packet retransmissions and marks packets to perform traffic failover as discussed in Algorithm 1. The total software modules are implemented in 1066 lines of C code. The RePaC library is a lightweight software module that can be easily installed on servers.

We evaluate the performance of RePaC in a fat-tree topology as shown in Figure 8. In this testbed, all the links are 25Gbps. 8 spine switches (Broadcom Trident 3) interconnect 4 leaf switches (Broadcom Tomahawk 3). We run SONiC [35] (an open-source network operating system) on all switches and adopt different ECMP configurations at each tier to avoid traffic polarization. Four physical servers are connected to leaf switches L1 & L2, and another four are connected to L3 & L4. With each server connected to two switches, the server can still connect to another leaf switch if one fails. Each server is equipped with Linux 3.10 or Linux 4.19 and TCP NewReno. These settings are consistent with our production data center networks.

5.2 Effectiveness and Overhead

Path control upon link and device failures: We first evaluate the accuracy of path control under various failure scenarios. Link and device failures are injected based on failure statistics collected in production data centers during one week. We gauge the accuracy of path control by calculating the percentage that the selected path is the expected path. We test with various numbers of spine switches to validate the perfor-

Table 4: Accuracy of path control under failures

# of spine switches	4	8	16	32
RePaC w\o failures	100.0%	100.0%	100.0%	100.0%
RePaC w\ failures	97.0%	98.5%	99.24%	99.62%
Random w\o failures	25%	12.5%	6.3%	3.1%

mance under different network scales. As shown in Table 4, the accuracy increases with the number of spine switches since the proportion of failed links of total links decreases. As discusses in Section 3.1, RePaC can provide accurate path control for the first link failure if RePaC is triggered before BGP changes the bucket. But RePaC might not work at the first attempt if there are two and more link failures in the same ECMP group. Table 4 shows RePaC provides >97.0% accuracy while the random path selection can only provide <25% .

Overhead: We test the CPU overhead and the memory cost of RePaC with 1 thousand to 1 million concurrent connections. The average CPU cost on a 64-core processor is about 0.01% for 1k connections and 0.05% for 1m connections. Note that the performance of RePaC is independent of the kernel version because RePaC is lightweight and implemented in a portable VNIC module. At each server, we allocate about 64 bytes of extra memory for each connection to keep track of its state. We also evaluate whether RePaC reduces packet processing speed. RePaC looks up the flow table on every packet, checks packet sequence number, and updates the flow state accordingly. These bring a few extra memory accesses to the data path. We observe no degradation in packet processing speed with different workloads.

5.3 RePaC for Failover Evaluation

Experiment methodology: We study the failover performance of RePaC by measuring throughput and downtime under common failure scenarios. We run a high-priority application on the VNIC interface to collect throughput measurements at a 20ms interval, which also helps us determine the downtime upon failure. We consider two failure scenarios, silent drop failures and link failures. We simulate silent drop failures by configuring an ACL rule to discard the data traffic. We simulate link failures by issuing `ifconfig down` command to deactivate one of the interfaces from receiving and sending data from switch CLI. There are many existing studies on failover [19, 23, 43, 44, 50]. Among them, MPTCP is a fair baseline as an end-host-based mechanism. MPTCP employs a group of subflows with random paths for failover without adding any centralized controller or advanced hardware. We use MPTCP v0.90 with the redundant scheduler for comparison with RePaC. We test MPTCP with the redundant scheduler over the default scheduler since it provides better fault tolerance by sending redundant copies through all subflows.

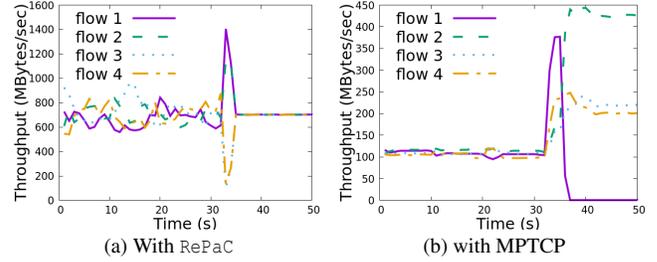


Figure 9: Throughput during failure

Failure recovery performance: We compare RePaC with both end-to-end and in-network failover design. Compared with end-to-end failover MPTCP, which recovers with a random backup path, RePaC provides better recoverability. Compared with in-network failover approaches, RePaC covers more failure types and reduces downtime by orders of magnitude.

We first compare RePaC with MPTCP in Figure 9. Figure 9a shows an example trace the silent drop on leaf switch L1 get recovered after silent drop failure with RePaC. Flow 3 and flow 4 are forwarded to L1 by ECMP and hence dropped by ACL. Both flows experience around 1 second of disruption and then are re-routed to switch L2. Flow 1 and flow 2 forwarded by switch L2 only experience oscillation. During the oscillation, the throughput of flow 1 and flow 2 first doubles when the other two flows stop transmission, then returns to similar throughput as before. All flows eventually share the bandwidth of the bottleneck link fairly with congestion control. Note that the bottleneck is the L3-E5 link, so the throughput before and after the failover are similar. In comparison, Figure 9b shows that MPTCP failover performance is unsatisfactory. We configure each flow with 4 subflows to increase the path diversity. Figure 9b shows that MPTCP can recover all affected flows except flow 1 because all subflows of flow 1 are dropped. The throughput of flows 2-4 increases after flow 1 is deactivated. Overall throughput performance with MPTCP is worse than RePaC since MPTCP with the redundant scheduler copies packets on all available subflows.

We further compare the failure recovery ratio between RePaC and MPTCP with different numbers of subflows for MPTCP. We define the recovery ratio as the probability that a flow affected by failure is recovered. As shown in Figure 10, MPTCP offers a higher recovery ratio with more subflows. Although MPTCP provides a reasonable recovery ratio with the redundant scheduler, RePaC performs better by leveraging path diversity with the pathmap. RePaC outperforms MPTCP with 2(4) subflows by 36%(21%). RePaC outperforms MPTCP because MPTCP cannot guarantee there exists at least a subflow unaffected by the failure.

We also compare RePaC with in-network failover mechanisms under various failure scenarios. As shown in Figure 11a, we start a single TCP flow from E1 to E5 and measure the downtime after simulating a device failure at L1. We simulate the device failure by deactivating all the interfaces. In prac-

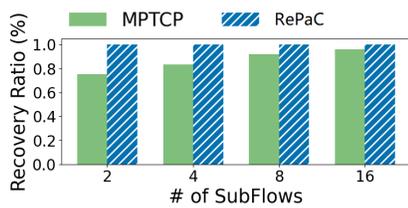


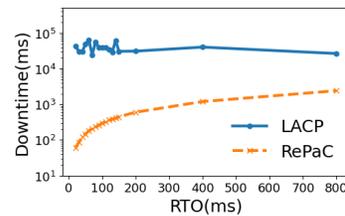
Figure 10: Recoverability comparison between RePaC and MPTCP.

tice, device failures might be triggered by hardware failure, unexpected reboot, or power loss. Although this type of failure can be detected by Link Aggregation Control Protocol (LACP) using heartbeat signals, it takes up to 90 seconds. In comparison, RePaC significantly reduces the downtime by orders of magnitude. Note the recovery Algorithm 1 initiates path control after the second TCP retransmission. Our solution performs better with a smaller minimum RTO. RePaC reacts to failures in around 60ms with a 20ms minimum RTO. Theoretically, RePaC can provide sub-ms recovery if the end hosts adopt high-resolution timers.

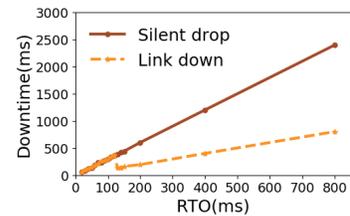
In Figure 11b, we evaluate how our solution can cooperate with the existing in-network failover approaches used in data centers, such as link scan, Bidirectional Forwarding Detection (BFD), LACP, and BGP. We compare the downtime caused by two types of failures, silent drop failures and link down events. For silent drop failures, our design can recover affected flows within three times of RTO, as shown in the solid line in Figure 11b. After a link down failure is injected, RePaC is triggered faster than in-network approaches when the second retransmission is triggered before the shortest in-network failover timers (130 ms for the link scan function). Note RePaC kicks in after around the triple of the minimum RTO. When the minimum RTO increases, RePaC will not be triggered since the link scan function would recover failures faster. In summary, Figure 11 proves RePaC can cooperate well with existing failover mechanisms and help to reduce the flow downtime.

5.4 RePaC for Traffic Engineering Evaluation

Experiment methodology: We gauge the performance of load balancing and flow completion time of RePaC compared with random path selection, precise path control via XPath [21], and in-network load balancing via CONGA [3]. We evaluate RePaC for load balancing under two scenarios: one-to-one flows and many-to-one incast flows. For one-to-one flows, we vary the *src_port* to vary the path selection of these parallel flows. For many-to-one incast flows, we vary *src_ip* and *src_port* for four servers under the same rack. We assume the source port number is flexible for traffic engineering applications. For many-to-one flows, we assume the source IP address is also assignable. We vary the least signifi-



(a) Comparison between RePaC and in-network failover under device failure



(b) RePaC with in-network failover under silent drop and link down

Figure 11: Downtime comparison under various minimum RTOs

cant 8 bits of *src_ip* inside a cluster to improve path diversity. We test with three representative traffic patterns, same-size flows, webserver flows, and Hadoop flows. Webserver flows follow a uniform size distribution, and Hadoop flows follow a skewed size distribution [40].

Load balance performance: In Figure 12, we compare the bandwidth utilization of RePaC path planning with random path planning. The bandwidth utilization is measured using the average utilization ratio of the links in the network. For one-to-one flows, as seen in Figure 12a, RePaC increases the bandwidth utilization by up to 3 times when the total number of flows is 16. For many-to-one connection, Figure 12b shows the bandwidth utilization of links when flows of the same size from different sources are sent to a single destination. Our solution can evenly distribute the flows across different paths, achieving well-balanced bandwidth utilization. For the webserver case, as shown in Figure 12c, the bandwidth utilization increases from 0.54 to 0.99 using RePaC, compared to 0.34 to 0.85 with random selection as the number of flows increases from 16 to 4096. Figure 12d shows the bandwidth utilization for Hadoop with skewed flow size distribution. Both algorithms show low bandwidth utilization when the number of flows is small. When more flows join, RePaC can 20% to 50% better bandwidth utilization.

In Figure 13, we compare our solution with random path selection by measuring the load on each path when the number of concurrent flows increases from 16 up to 4096. Here we calculate the load at each path and plot the standard deviation over mean in Figure 13. As the number of flows increases, RePaC provides much better performance. For webserver traffic, the std./mean reduce by 58% to 97% as shown in Figure 13a. Our design tends to select a path with the least load for each flow. In Figure 13b, when the number of flows is greater than 64, the std./mean is reduced by more than 41%.

For time-sensitive applications, such as distributed file systems (DFS), flow completion time is crucial. We simulate DFS by sending 100GB of data from one host to another. We compare the completion time of all concurrent flows between RePaC and random path selection. RePaC can plan paths based on the traffic load, so elephant flows are distributed to separate paths. We notice that the average completion time of our solution is considerably less than the random path selection as shown in Figure 14a. When there are 16 flows, the comple-

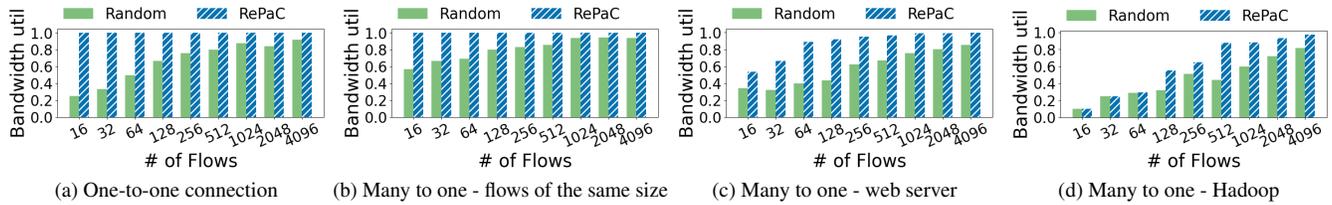


Figure 12: Link bandwidth utilization of path planning.

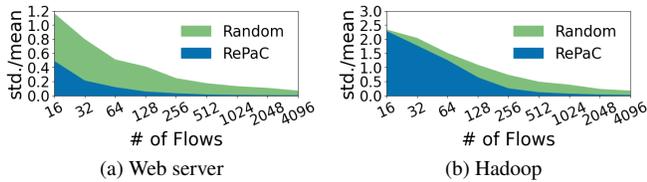


Figure 13: Load balance of path planning. The shading area shows the std./mean. of load on each switch. Larger shading area means load is more unbalanced.

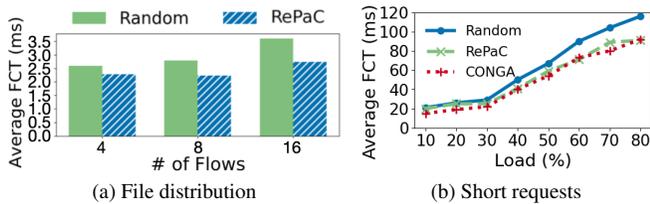


Figure 14: Flow completion time.

tion time reduces by 25% from 3.6 seconds to 2.7 seconds. Besides large file distribution, we also simulate short requests and gauge the flow completion time compared with random path selection and CONGA [3]. We use simulation since CONGA requires advanced hardware. Each host will inject flows with empirical traffic distributions of short requests into different queues based on the given path selection algorithm. CONGA will measure the queuing time of each path and choose the path with the least queuing time to send flowlets, thus performs best. RePaC tends to select the path with the least number of concurrent flows. As shown in Figure 14b, RePaC’s performance is comparable with CONGA.

Load balance performance under topology updates: The network topology might change if network devices are removed for maintenance or upgrade. In Figure 15, we compare the load balance performance between random path selection, RePaC, and XPath [21] when the network topology changes. Since XPath is based on source routing, which limits its deployment on our testbed, we use simulation with the same testbed topology. We simulate multiple concurrent flows between end-hosts and compare load balance by the standard deviation of the load on each available path. XPath relies on a centralized controller to perform topology updates, which, we assume, can provide accurate path estimation. As shown in Figure 15a, without topology updates, both RePaC and XPath distribute the flows evenly. After removing a spine

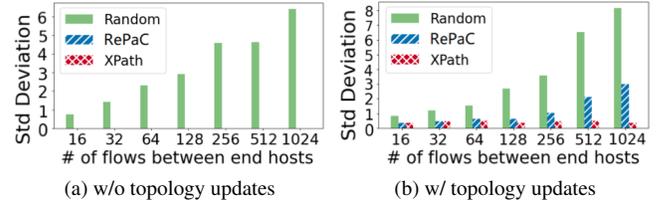


Figure 15: Load balance performance when topology changes

switch, as shown in Figure 15b, XPath can still balance the flows using accurate path information. RePaC can distribute most flows evenly on the unchanged paths using outdated pathmaps. In conclusion, XPath has the best performance leveraging pre-installed routes and real-time path information updates. RePaC degrades because of the outdated pathmap. However, both outperform the random path selection significantly. Compared with XPath, RePaC is a lightweight solution that significantly outperforms the random scheme without redesigning the routing protocols .

6 Discussion

Our work is the first study that performs a detailed analysis on the ECMP hashing linearity and its applications to path control in data centers. More work could be done to further explore the design of multi-path hashing mechanisms and their implications to data center networks.

ECMP configuration optimization: Our analysis has demonstrated that the properties of ECMP hashing algorithms have significant impacts on traffic engineering and failover in data center networks. However, ECMP configurations in today’s production data centers are still done by operators using ad-hoc approaches. Future investigations of ECMP might reveal other interesting properties rather than linearity, which could facilitate the data center network optimization and result in a more scientific approach to network-wide configuration optimization.

Flow analysis based on hash simulation: Analysis of the ECMP path selection algorithm also sheds insights on how flows are distributed. Using real-time network topology information, network operators could build a lightweight simulation network with the model of ECMP hashing behaviors. The simulation network can answer many what-if questions about the network performance and flow distributions, such as how many flows will be affected by certain failure events

or a congested link.

Programmable hashing on switch ASICs: Today’s data center switch ASICs provide limited interfaces to configure ECMP hash algorithms and parameters. Even on programmable switch ASICs (e.g. Tofino), where users can program the packet processing pipeline with network programming languages, there is still limited programmability to define ECMP hashing behaviors. Given that the hash algorithm could significantly impact the overall networking performance, a programmable interface that allows users to redesign the hash algorithms could lead to more innovations in this space.

RePaC capable applications: RePaC provides a lightweight mechanism for applications to predict and control the paths of packet flows in the network, without the requirements for deploying any new protocols, centralized controllers, or advanced hardware. Many applications can benefit from path diversity. For example, network probing tools, such as PingMesh [17], can guarantee full network coverage while minimizing the probing overhead with the help of pathmap. The same benefits apply to applications with low latency and high throughput requirements, such as distributed file systems or AI training clusters. RePaC provides a powerful tool for the applications to schedule flows to avoid traffic congestion and disruption.

RePaC and future ECMP: RePaC requires hashing linearity, which is a built-in feature for linear hashing algorithms. For future hashing algorithms, there is no conclusion on whether linear hashing is preferred or non-linear hashing. Existing ASICs combine XOR and CRC to improve hashing while keeping linearity [33]. We are not aware of any studies on the possible tradeoff between linearity and hashing performance. It is still unclear whether non-linear hashing definitely outperforms all variants of linear hashing. Lightweight path control for future variants of hashing algorithms remains an interesting research problem. In order to support RePaC, ECMP can only adopt linear hashing algorithms. We expect future standardizations of ECMP to include linearity as another metric to consider besides hashing performance.

7 Related Work

RePaC is related to several prior lines of work:

End-host based path control: There are two spectrums of end-host-based path control. The first spectrum of approaches gets rid of ECMP and redesigns the routing protocols in the network [21, 22, 39, 41]. XPath is the most recent work, which identifies and pre-installs routes on switches [21]. Though XPath reduces the routing table storage with compression algorithms, the communication between end-hosts and the path manager under topology changes and link failures incurs too much overhead. MPLS is the common practice for traffic engineering in core networks [39, 41]. However, MPLS is not suitable for data centers since it can only support a limited

number of tunnels. The OpenFlow-based solution relies on on-chip forwarding rules and compresses forwarding rules with a tiny flow table [22]. However, this approach does not apply to data centers due to too many flows. Fundamentally, these approaches require excessive routing/forwarding tables because of the large scale of header space and the variability of paths.

Another spectrum still keeps ECMP as the path selection in switches but extracts the path selection model. Volur [47] attempts to provide path control with a centralized controller that aggregates routing information from in-network switches and disseminates routing information to end-hosts. However, the path selection model incurs too much overhead since it uses a mapping from the entire header space to all possible paths. The header space grows exponentially to the number of bits in header fields used by ECMP. We argue that explicit path control is not necessary so that we can reduce the complexity of the path selection model. Compared with this spectrum of work, we leverage the linearity of ECMP path selection and reduce the model complexity significantly.

Failover and traffic engineering: While our work is closely related to prior studies on failover [14, 19, 23, 25, 43, 44, 50] and traffic engineering [2, 3, 5], it is also fundamentally different. First, RePaC enables path control, which is different from the random path selection scheme in [19, 23, 44]. Second, RePaC is lightweight and easy to deploy, while existing approaches rely on advanced programmable switches [3, 14, 19, 43, 44, 50], switch redesigning (e.g. installing Openflow) [5], or a centralized scheduler to get path information [2]. Third, RePaC enables data traffic based path discovery, which outperforms traceroute traffic based approach [25] since the header fields in original data traffic and traceroute traffic are different.

8 Conclusion

In this paper, we analyze the ECMP hash algorithms used in today’s data center switch ASICs. Our analysis shows that the hash algorithms (e.g. XOR and CRC) used in the most popular switch ASICs on the market maintain linearity. We analyze how linearity sheds insights on relative path control. We design RePaC to leverage linear property and show that RePaC can be used in two representative applications, faster failover and traffic engineering. Through extensive evaluation in production data centers, we show that RePaC is easy to deploy and benefits failover and traffic engineering.

Acknowledgments

We would like to thank our shepherd, Amy Ousterhout, and the anonymous reviewers for helping us improve this paper. We would also like to thank the anonymous referees for their valuable comments and helpful suggestions on earlier versions of this paper.

References

- [1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 38(4):63–74, 2008.
- [2] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. Hedera: dynamic flow scheduling for data center networks. In *Nsdi*, volume 10, pages 89–92, 2010.
- [3] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 503–514, 2014.
- [4] Arista. Port channels and lacp. <https://www.arista.com/assets/data/pdf/user-manual/um-eos/Chapters/Port%20Channels%20and%20LACP.pdf>, 2019.
- [5] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, New York, NY, USA, 2011. Association for Computing Machinery.
- [6] Broadcom. Broadcom-network-switching-software. https://github.com/Broadcom-Network-Switching-Software/SDKLT/blob/7a5389c6e0dfe7546234d2dfe9311b92b1973e7b/src/bcmptm/include/bcmptm/bcmptm_rm_hash_internal.h, 2019.
- [7] Emily Carr. Why merchant silicon is taking over the data center network market. <https://www.datacenterknowledge.com/networks/why-merchant-silicon-taking-over-data-center-network-market>, 2019.
- [8] Guo Chen, Yuanwei Lu, Yuan Meng, Bojie Li, Kun Tan, Dan Pei, Peng Cheng, Layong (Larry) Luo, Yongqiang Xiong, Xiaoliang Wang, and Youjian Zhao. Fast and cautious: Leveraging multi-path diversity for transport loss recovery in data centers. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 29–42, Denver, CO, June 2016. USENIX Association.
- [9] Cisco. Understanding etherchannel load balancing and redundancy on catalyst switches. <https://www.cisco.com/c/en/us/support/docs/lan-switching/etherchannel/12023-4.html>, 2007.
- [10] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.
- [11] M. Davies. Traffic distribution techniques utilizing initial and scrambled hash values, 2010. US Patent US7821925.
- [12] Dell. Dell configuration guide for the s4048-on system 9.9(0.0) ecmp-rtag7. https://www.dell.com/support/manuals/us/en/19/force10-s4048-on/s4048_on_9.9.0.0_config_pub-v1/rtag7?guid=guid-9bda04b4-e966-43c7-a8cf-f01e7ce600f4&lang=en-us, 2019.
- [13] Nathan Farrington and Alexey Andreyev. Facebook’s data center network architecture. In *2013 Optical Interconnects Conference*, pages 49–50. Citeseer, 2013.
- [14] Soudeh Ghorbani, Zibin Yang, P Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Drill: Micro load balancing for low-latency data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 225–238, 2017.
- [15] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 350–361, 2011.
- [16] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: a high performance, server-centric network architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 63–74, 2009.
- [17] 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.
- [18] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based load balancing for fast datacenter networks. *ACM SIGCOMM Computer Communication Review*, 45(4):465–478, 2015.
- [19] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*,

- pages 161–176, Boston, MA, February 2019. USENIX Association.
- [20] C. Hopps. Analysis of an equal-cost multi-path algorithm. RFC 2992, RFC Editor, November 2000.
- [21] Shuihai Hu, Kai Chen, Haitao Wu, Wei Bai, Chang Lan, Hao Wang, Hongze Zhao, and Chuanxiong Guo. Explicit path control in commodity data centers: Design and applications. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 15–28, Oakland, CA, May 2015. USENIX Association.
- [22] Sangeetha Abdu Jyothi, Mo Dong, and P. Brighten Godfrey. Towards a flexible data center fabric with source routing. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15*, New York, NY, USA, 2015. Association for Computing Machinery.
- [23] Abdul Kabbani, Balajee Vamanan, Jahangir Hasan, and Fabien Duchene. Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies, CoNEXT '14*, page 149–160, New York, NY, USA, 2014. Association for Computing Machinery.
- [24] M. Kalkunte. High speed trunking in a network device, 2005. US Patent US20060114876A1.
- [25] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. Clove: Congestion-aware load balancing at the virtual edge. In *Proceedings of the 13th International Conference on Emerging Networking Experiments and Technologies, CoNEXT '17*, page 323–335, New York, NY, USA, 2017. Association for Computing Machinery.
- [26] Parantap Lahiri, George Chen, Petr Lapukhov, Edet Nkposong, Dave Maltx, Robert Toomey, and Lihua Yuan. Routing design for large scale data centers. *NANOG 55*, 2012.
- [27] Petr Lapukhov, Ariff Premji, and Jon Mitchell. Use of bgp for routing in large-scale data centers. *Internet Requests for Comments RFC Editor RFC*, 7938, 2016.
- [28] Guohan Lu. Sai hash enhancement with user defined field. <https://github.com/opencomputeproject/SAI/blob/master/doc/SAI-Proposal-13-Hash-UDF.md>, 2016.
- [29] Guohan Lu and Xin Liu. Sai update and look forward. <https://www.opencompute.org/files/OC P2018-SAI-Engineering-Talk-MSFT-final.pdf>, 2018.
- [30] 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, 2018.
- [31] Brad Matthews and Puneet Agarwal. Flow based path selection randomization, 2013. US Patent US8503456.
- [32] Liron Mula, Gil Levy, and Aviv Kfir. Using consistent hashing for ecmp routing, 2017. US Patent US9853900B1.
- [33] Cumulus networks. Equal cost multipath load sharing - hardware ecmp. <https://docs.cumulusnetworks.com/cumulus-linux/Layer-3/Equal-Cost-Multipath-Load-Sharing-Hardware-ECMP>, 2019.
- [34] opencomputerproject. Sai. <https://github.com/opencomputeproject/SAI/blob/484b8b150e53b9a818af9a850d4a78581ca7e9bc/inc/saiswitch.h#L188>, 2019.
- [35] opencomputerproject. Sonic. <https://azure.github.io/SONiC/>, 2019.
- [36] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. In *Proceedings of the ACM SIGCOMM 2011 Conference, SIGCOMM '11*, pages 266–277, New York, NY, USA, 2011. ACM.
- [37] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath TCP. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 399–412, San Jose, CA, April 2012. USENIX Association.
- [38] Jarno Rajahalme. Performing a finishing operation to improve the quality of a resulting hash, 2014. US Patent US10193806B2.
- [39] Eric Rosen, Arun Viswanathan, Ross Callon, et al. Multiprotocol label switching architecture. 2001.
- [40] 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, SIGCOMM '15*, pages 123–137, New York, NY, USA, 2015. ACM.
- [41] Martin Suchara, Dahai Xu, Robert Doverspike, David Johnson, and Jennifer Rexford. Network architecture

- for joint failure recovery and traffic engineering. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '11, page 97–108, New York, NY, USA, 2011. Association for Computing Machinery.
- [42] Kovsky T., J. Tsai, and Joe Chang. High performance crc calculation method and system with a matrix transformation strategy, 2003. US Patent US7219293B2.
- [43] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with switchpointer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 453–456, Renton, WA, April 2018. USENIX Association.
- [44] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 407–420, Boston, MA, March 2017. USENIX Association.
- [45] WiKipedia. Computation of cyclic redundancy checks. https://en.wikipedia.org/wiki/Computation_of_cyclic_redundancy_checks, 2019.
- [46] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. Resilient datacenter load balancing in the wild. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 253–266, 2017.
- [47] Qiao Zhang, Danyang Zhuo, Vincent Liu, Petr Lapukhov, Simon Peter, Arvind Krishnamurthy, and Thomas E. Anderson. Volur: Concurrent edge/core route control in data center networks. *CoRR*, abs/1804.06945, 2018.
- [48] Zhehui Zhang, Haiyang Zheng, Jiayao Hu, Xiangning Yu, Chenchen Qi, Xuemei Shi, and Guohui Wang. Supplementary material to hashing linearity enables relative path control in data centers. https://zhehuizhang.github.io/files/atc21_sup.pdf, 2021.
- [49] Zhiruo Cao, Zheng Wang, and E. Zegura. Performance of hashing-based schemes for internet load balancing. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, volume 1, pages 332–341 vol.1, March 2000.
- [50] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 362–375, 2017.

Live in the Express Lane

Patrick Jahnke^{*†} Vincent Riesop[†] Pierre-Louis Roman[‡] Pavel Chuprikov[‡] Patrick Eugster^{‡*§}
^{*}TU Darmstadt [†]SAP [‡]Università della Svizzera italiana [§]Purdue University

Abstract

We introduce Express-Lane (X-Lane), a novel system for mitigating interference in data center infrastructure to improve the liveness of coordination services. X-Lane follows a novel design from the ground up to achieve interactions with ultra-low latency in the single-digit microsecond range and jitter in the nanosecond range, while the remaining interaction is treated as usual. To show X-Lane’s applicability and generality we implemented and evaluated two services atop it on commodity hardware in a production environment of SAP SE: a failure detector (X-FD) with detection time under 10 μ s and a Raft implementation (X-Raft) with latencies under 20 μ s. We further show the smooth integrability of X-Lane services by replacing the replication protocol of Redis with X-Raft, making it strongly consistent while improving latency 18 \times and write throughput 1.5 \times .

1 Introduction

In the last decade, a tremendous increase in Internet connectivity and the need for more computational performance changed the way we conceive applications. Today, most new applications are conceived as distributed, and in particular cloud-based, applications. The design of data centers and middleware layers then has to take into account all requirements for distributed coordination, including performance, fault-tolerance, and consistency [16] — a hard task.

Interference in distributed systems. Most distributed system designs treat the underlying infrastructure as a generic communication system. One of the main issues with this abstraction is the longstanding problem of interference of concurrent interactions and thus unpredictable latency of commodity networks and hosts [19]. Many distributed systems suffer from jitter induced by interference, manifesting through packets that may be arbitrarily delayed in the network (as well as reordered or dropped), and unbounded processing times.

Many applications and components have been designed to cope with the unpredictability of the infrastructure by making weak synchrony assumptions to guarantee a safe execution of their protocol. Yet, they rely on upper bounds for the latency of their interactions to ensure liveness, by way of timeouts, and as thus benefit strongly from interactions with low latency and bounded jitter. This is especially the case for coordination tasks [52] whose use is widespread in practice. Types of systems using the ZooKeeper [28] coordination service based on the popular Paxos [37] protocol by default or as option for coordination/fault tolerance include resource management (e.g., Mesos [25], YARN [54]), key-value and wide-column stores (e.g., Accumulo [20], HBase [1], etcd [4], TiKV [10]), data analytics (e.g., Hadoop [12], Spark [58]), or distributed filesystems (e.g., HDFS [13]) to only name few.

X-Lane. The research question underlying this work is whether interference in data center commodity systems can be mitigated to greatly accelerate coordination tasks lying at the core of distributed systems.

Prior works on low latency communication (e.g., [14,24,41,44,48,50]) focus on reducing 99th percentile latency where packets may be sacrificed (dropped) to maintain a good performance in most cases (e.g., fitting a given service-level objective (SLO) for 99% of the packets). Our goal is to address not only fast but also *timely sensitive interactions* for tasks that exhibit severe performance degradation upon delayed message delivery (e.g., when timeouts trigger). To this end, we aim at reducing maximum jitter to a point where it becomes so small relative to an already very low latency, that, in practice, it can be assumed to be bounded. Moreover, we include *endhost response times*, and only provide bounded jitter to applications that rely on it (e.g., for coordination). Thus we introduce with Express-Lane — X-Lane for short — an interference-free environment for select interactions with ultra low latency in the *single-digit microsecond range* and bounded jitter in *nanosecond range*. The remaining interactions follow common design principles. While being more generic in design compared to prior work on minimizing av-

erage latency, and also considering endhosts, X-Lane delivers significantly tighter bounds for latency and jitter for commodity hardware (HW) and software (SW).

In short, X-Lane isolates and prioritizes packets traversing it by using traffic engineering techniques to provision and monitor resources dedicated for X-Lane, and by neutralizing sources of interference inherent to data center infrastructures, i.e., interference present in endhosts/servers, switches, and links. X-Lane strives first and foremost to minimize jitter, and in the process also achieves unprecedented low latency.

Contributions and roadmap. This paper contributes:

- §2 Design of X-Lane atop commodity HW and SW, and for intelligent network devices (smartNICs) when available;
- §3 Traffic engineering approach incorporating residual jitter and queuing delay to perform packet-level latency analysis in X-Lane;
- §4 Implementation of X-Lane overcoming interference causing jitter on top of commodity HW and SW, as well as improvements and simplifications taking advantage of Netronome’s NFP-4000-based smartNICs [8];
- §5 Definition and implementation of two example asynchronous services using X-Lane: a failure detector dubbed X-FD, and a state machine replication protocol dubbed X-Raft adapted from Raft [45];
- §6 Evaluation of X-Lane in a production data center of SAP SE through the deployment of the two services. We measure median latency and maximum jitter of X-Lane on commodity HW and SW (Linux) (5.130 μ s latency and 655 ns jitter) and smartNICs (4.133 μ s latency, 152 ns jitter) *with heavy concomitant traffic over the course of 21 days*. Further comparisons display vast improvements over DPDK [21] (1.735 \times lower latency, 81,816 \times lower jitter), and QJump [24] (1.501 \times lower latency, 72,758 \times lower jitter), which greatly affect the coordination of distributed systems. We also show the applicability of X-Lane by integrating X-Raft into the Redis key-value store [9], making it strongly consistent while decreasing latency 18 \times and increasing write throughput 1.5 \times .

We compare X-Lane to related work in §7 before we draw the conclusions and discuss future work in §8. Additional material can be found on the project website [30].

2 X-Lane Design Overview

With X-Lane we propose an explicit express lane for timely sensitive interactions, following our original design outlined in Fig. 1. X-Lane is isolated from the “regular system” which follows common design principles. This architecture is reminiscent of earlier models of separate systems [55, 56], yet realizes them concretely, in a single infrastructure, with commodity HW and SW.

2.1 Communication Model

X-Lane’s novelty is characterized by an explicit upper-bound on the latency of all the messages sent by a given process p to another process q , i.e., X-Lane keeps the latency of every such message within $[\lambda_{\min}^{p,q}, \lambda_{\min}^{p,q} + \delta_{\max}^{p,q}]$, where $\lambda_{\min}^{p,q}$ is the best-case latency, and $\delta_{\max}^{p,q}$ is its concomitant maximum jitter. In the following, we denote jitter δ as a deviation from the best-case latency λ_{\min} .

We achieve bounded communication latency in X-Lane by implementing a *periodic unicast protocol* where a process p can send a message to a given process q with latency upper bound $\lambda_{\min}^{p,q} + \delta_{\max}^{p,q}$, but under two constraints: p can send only once during every period $\pi^{p,q}$, and the packet size may not exceed $\sigma^{p,q}$. In addition, we specifically address one-to-many communication patterns by a *periodic multicast protocol* that allows a process p to send a message to a *set of processes* Q with a common latency range $[\lambda_{\min}^{p,Q}, \lambda_{\min}^{p,Q} + \delta_{\max}^{p,Q}]$. A crucial requirement for both our protocols is that *all their parameters become known by the sending process at the protocol setup time*, i.e., before the first use, in order to allow services to adjust their internal timeouts for the best possible performance.

Note, purely bandwidth-oriented communication abstractions are *not* suitable for X-Lane, for they leave message size unspecified, while, clearly, no latency bound would hold uniformly for *every* message size, and queuing behind an arbitrarily large message leads to unbounded maximum jitter.

X-Lane is able to provide timely sensitive interaction that exhibits stable behavior as long as interconnecting devices function properly. Hence X-Lane is best used to improve the liveness of coordination tasks that assume an asynchronous communication model to guarantee safety properties.

Timely unicast and multicast serve as *backbone for all communication* between processes in X-Lane. In the following, “periodic protocol” refers to “unicast protocol or multicast protocol”. Bounding latency in the sending process is addressed in §2.4 and detailed in §4.

2.2 Components Overview

To achieve the properties provided by the two periodic protocols, X-Lane introduces a software-defined networking (SDN) controller that takes on two main orchestration responsibilities: 1) *resource allocation*, i.e., answering requests from services with the most suitable protocol parameters, subject to network capacity constraints; and 2) *resource tuning*, i.e., keeping overall usage of X-Lane low. Traffic engineering (TE) techniques that underpin this operation are presented in §3.

The X-Lane controller interacts with each endhost via a client integrated in the X-Lane (Linux) kernel module (X-KM) loaded on each endhost. The client exposes the controller API (cf. List. 1) to services forwarding requests and responses in both directions. It is important to note that only the bounded communication over X-Lane is managed by the X-Lane con-

```

// Service request parameters for X-Lane resources
struct request {
    int loadsize;           // max packet size (B)
    int period;            // packet period (μs)
    struct {
        uint32_t ip;       // MCast or UCast IPv4
        uint16_t port;     // service port
    } receiver;
};

// Resources approved by the X-Lane controller
static const int UNBOUNDED = -1;
struct resources {
    int loadsize;           // max packet size (B)
    int period;            // approved period (μs)
    int minLatency;        // minimum latency (ns)
    int maxJitter;         // maximum jitter (ns)
};

// Reason for resource modification
enum Reason { TE, BW_EXCEEDED, BW_UNUSED };
// Downcalls from services to controller
↓ resources requestBandwidth(request req);
↓ void releaseBandwidth();
↓ void changeBandwidth(request req);
// Upcalls from controller to applications
↑ void bandwidthChanged(resources res,
    Reason reason = TE);
↑ void bandwidthTerminated();

```

List. 1: Extract of the X-Lane controller C API used for resource allocation and tuning. Structure `resources` defines a timely periodic protocol. The first three methods are called by services the next two are upcalls/callbacks.

troller. The rest of the communication proceeds as usual and uses the remaining resources in the usual best-effort manner. If no requests are ever made to the X-Lane controller, no network resources are spared or lost.

2.3 Overview of Jitter Sources

To implement an express lane usable in practice for time-sensitive tasks, we need to mitigate the inherent interferences in data center computing. We expose and address numerous jitter sources in §4. In short, we identify the following causes:

- **Packet loss:** Packets can be lost, leading to retransmissions and thus uncertain latency. Besides intentional drops (e.g., for security), packet loss has two well-known causes:
 - **Bit flip errors:** Bits can get flipped in links, leading to packets being marked as corrupted and discarded (§4.1);
 - **Buffer overflows:** Packets are dropped when the finite resources on processing units are overloaded (§4.2).
- **Intrinsic jitter:** While commodity switching devices forward packets with little jitter (§4.2), endhosts and their commodity components have been becoming more complex, leading to many sources of jitter (§4.3) and motivating the need for moving the intelligence closer to network devices (§4.4). The lack of bounds on jitter further makes packet delay hard to distinguish from packet loss.

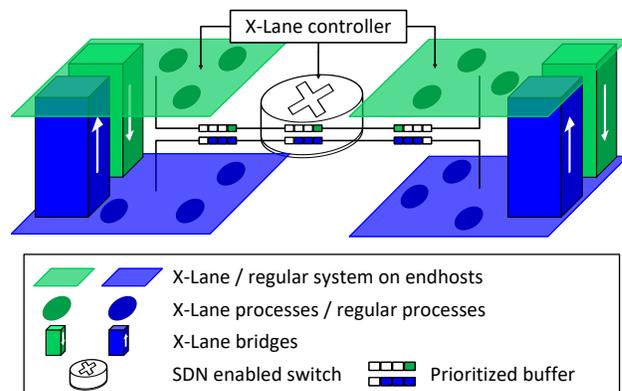


Figure 1: Separating the traffic of Express-Lane (X-Lane) and regular communication on switches to prioritize packets and prevent losses in the former. An SDN controller sets switches' rules to adapt buffer allocation and processing priority. X-Lane is interfaced to the regular system via its bridges.

2.4 X-Lane (Based) Services

X-Lane enables processes executing on the regular system to interact with the X-Lane services that may offer *timely* responses thanks to the unique timing properties of communications of X-Lane. There are a few intricacies to X-Lane that developers must take into account when interacting with and/or developing these services. First, applications and services, being in separate lanes, must use a specific interface to exchange data with each other. Second, X-Lane handles communications differently than in the regular system.

Building bridges between lanes. On endhosts, services must communicate with applications which have to deal with shared processor time. This resource sharing introduces unpredictable jitter for those processes while critical interactions need an upper bound for certain tasks. Hence X-Lane provides two sets of queues, called *bridges*, to establish the interface between processes in X-Lane and on the regular system.

Fig. 1 depicts the bridges. The express-to-regular (X-R) bridge (green cuboid) grants write access to X-Lane (green parallelogram) and read access to the regular system (blue parallelogram); inversely for the R-X bridge (blue cuboid).

Bridges are addressable using direct memory access (DMA) over PCIe (to minimize jitter, cf. §4.3) but are placed at different locations depending on the endhost HW configuration.

Using X-Lane services. Services are implemented as components of the X-KM (cf. §5 for already available services), and as thus have direct access to the client of the controller and to the network interface card (NIC) bridge, another X-KM component responsible for communication with the NIC. Each service has a dedicated queue in the R-X bridge where it

can receive (1) queries from applications wishing to start/stop using that service, and (2) queries and payloads specific to that service API. When an application starts using a service, the service requests network resources from the X-Lane controller and spawns a new queue in the X-R bridge dedicated to messages from this service to that application. The NIC bridge bundles up all the payloads from a service into packets and sends them over the wire at the allowed periodicity (cf. period in [List. 1](#)), and unpacks payloads on the receiver side. Like drivers, the bridge implementation varies between HWs.

Express communication on commodity HW. While commodity NICs rapidly process and copy packets to the main memory, they are not programmable. Procedures to send and receive packets must thus be executed by the CPU.

When handling packets that belong to X-Lane, guaranteeing minimal response time and tight timing bounds for these procedures is especially challenging on commodity HW. There is an abundance of sources of jitter within the CPU itself and in the communication path between the CPU and the NIC that prevents a jitter-free streamline flow of packets. As a response, we implemented a series of countermeasures to enable X-Lane on commodity HW, greatly improving the time bounds over the regular system, as detailed in [§4.3](#). On commodity HW, all bridges are in the main memory.

Express communication on smartNICs. Unlike commodity NICs, new generation NICs — so-called smartNICs — are highly programmable. Tasks can be offloaded from the CPU to the processing engine of a smartNIC, ranging from packet pre-processing to complex programs. The (relative) simplicity of the HW and SW stacks of smartNICs, over those of an endhost operated by a Linux kernel, and their proximity to the physical interface enable for packets to be processed on smartNICs with far lower latency and jitter (cf. [§6.2](#)). This makes smartNICs ideal to handle X-Lane services.

Processing for sending and receiving packets over X-Lane is confined within the smartNIC. This processing is mostly as with commodity HW, but with direct access to the packet processing pipeline and the ingress and egress buffers on the NIC (cf. [§4.4](#)). The X-R bridge is stored in the smartNIC’s memory while the R-X bridge is in the endhost main memory.

3 Traffic Engineering for Tunnel Trees

The key underlying mechanism of the controller are *latency-bounded fixed-bandwidth* tunnels, more precisely — tunnel trees (due to multicast), from sender to receiver processes.

3.1 Tunnel Allocation Model

The X-Lane controller relies on SDN for tunnel setup. In particular, by acting as an SDN controller, it gets access to

network-wide view in a form of a *network topology graph* G and the means to manage switches. For every link $(u, v) \in G$, the following information is used: bandwidth $\text{bw}(u, v)$, size of an egress queue $\text{qlen}(u, v)$, minimum delay $\lambda_{\min}(u, v)$, and maximum jitter $\delta_{\max}(u, v)$. Importantly, $\lambda_{\min}(u, v)$ and $\delta_{\max}(u, v)$ need only include processing and propagation delays, which are stable for switches and are made stable at endhosts by X-Lane’s endhost implementation (see [§4](#)).

A resource *allocation* is represented by a set \mathcal{T} of *tunnels*, where every $T \in \mathcal{T}$ is a *directed subtree* of the topology graph G with a sender source $\text{snd}(T)$ and a set of receiver sinks $\text{rcvs}(T)$. Tunnels are in one-to-one correspondence with allocated resources shown in [List. 1](#); hence, for every $T \in \mathcal{T}$, we have packet size $\sigma(T)$, period $\pi(T)$, minimum latency $\lambda_{\min}(T)$, and maximum jitter $\delta_{\max}(T)$. X-Lane further employs TE techniques [[26](#), [27](#), [31](#)] to guarantee channel availability. The particular TE algorithm used for X-Lane is close to B4’s state-of-the-art approach [[27](#)] (with worst-case estimation of available throughput) but is built upon a finer-grained network model to allow for packet-level latency bounds.

The X-Lane controller does not make any explicit resource reservations in the network but instead relies on *rate limiting* at the endhosts, forcing services to adhere to periodic protocol parameters. Thus, the traffic for a given tunnel T consists of packets of size $\sigma(T)$ entering node $\text{snd}(T)$ precisely every $\pi(T)$ with starting time chosen arbitrarily for each T . Once a packet p from T arrives at a node u , p is either delivered, if $u \in \text{rcvs}(T)$, or p is placed into u ’s egress queue(s) corresponding to next hop(s) in T , provided there is sufficient buffer space, if not — p is dropped. Switching and/or processing delays at u are incorporated into latency and jitter of incoming links. At the egress queue, p waits for its turn to be transmitted according to FIFO order, and after $\text{size}(p)/\text{bw}(u, v)$ seconds more p leaves the queue. It takes anywhere between $\lambda_{\min}(u, v)$ and $\lambda_{\min}(u, v) + \delta_{\max}(u, v)$ before p enters the next hop v accounting for the minimum residual jitter remaining after applying techniques described in [§4](#).

TE of X-Lane accounts for both the intrinsic uncertainties of the system and uncertainties arising from multiple services sharing network resources. Ultimately, TE ensures that allocation \mathcal{T} is *valid w.r.t.* topology G , meaning that no actual system behavior violates $\lambda_{\min}(T)$ and $\delta_{\max}(T)$ for $T \in \mathcal{T}$.

3.2 Two-Phase Allocation Approach

Resources in X-Lane are allocated reactively, upon concrete requests by services.

To bootstrap a periodic protocol, a service calls the `requestBandwidth` method of the controller API passing the desired packet size and periodicity in a request structure r . The controller handles r as follows: 1) a new tunnel T is allocated between the sender and receiver(s); 2) switches’ meter tables are updated for resource monitoring; 3) parameter adjustments for other affected tunnels in \mathcal{T} are communi-

cated to corresponding services using the `bandwidthChanged` callback; 4) the approved resources with periodicity adjusted according to the allocation are returned to the service. Naturally, the new tunnel T must *match* the request r , i.e., packet size $\sigma(T)$ is equal to $r.loadsize$, $\text{snd}(T)$ is the process that originated r , $\text{rcvs}(T)$ correspond to $r.receiver.ip$, and $\pi(T) \geq r.period$ (mind the adjustment). The returned structure reflects all the T 's parameters of a periodic protocol (cf. §2.1): latency range $[\lambda_{\min}(T), \lambda_{\min}(T) + \delta_{\max}(T)]$, periodicity $\pi(T)$, and load size $\sigma(T)$. The service frees the resources by using `releaseBandwidth`. For the X-Lane properties to be reliable, every `bandwidthChanged` callback invoked by the controller comes with a grace period, during which the service can send messages under the old periodic protocol guarantees.

A distinguishing feature of our setting is the inevitable interference between already established tunnels and the new tunnel. Trying to minimize such interference, we arrive to an optimization problem underlying steps 1) and 3) above.

Problem (X-TE). *Given a network G , an allocation \mathcal{T} , and a sequence of service requests r_1, \dots, r_k , find a sequence of new tunnels $\mathcal{T}' = T'_1, \dots, T'_k$ and adjust parameters of \mathcal{T} , s.t., T'_i matches r_i for $1 \leq i \leq k$, $\mathcal{T} \cup \mathcal{T}'$ is valid w.r.t. G , and $\sum_{T \in \mathcal{T} \cup \mathcal{T}'} (\lambda_{\min}(T) + \delta_{\max}(T))$ is minimized.*

Solving X-TE directly is challenging as deriving parameters (or even checking validity) for a general \mathcal{T} is highly non-trivial due to interdependency between arrival times for packets queuing behind each other. Hence to simplify the problem, we split the allocation into two phases: *optimization* and *adjustment*. The *optimization* phase takes as input a request sequence and decides on the matching sequence of tunnels. In the current implementation, we allocate trees one-by-one; each tree is allocated incrementally by greedily attaching receiver sinks while minimizing the current value of the X-TE's objective function. The *adjustment* phase alters the parameters of all tunnels so they become valid w.r.t. the network G . Each tree is adjusted independently using depth-first search traversal calculating worst-case parameters.

3.3 Resource Monitoring and Tuning

In addition to its resource allocation task, the X-Lane controller improves resource utilization by monitoring and refining the set of already allocated tunnels.

Controller oversight. For instance, if a service wants to adjust its `loadsize` and/or `period` without disrupting other services, the `requestBandwidth` and `releaseBandwidth` methods force it to establish a new periodic protocol first, migrate all clients there, and only then release the old resources. This two-phase approach incurs artificial delay, adds complexity, and wastes X-Lane's resources. The `changeBandwidth` method of the controller's API shortcuts the process by leveraging the `bandwidthChange` mechanism discussed earlier.

When a service attempts to use more resources than assigned, some of its packets get dropped at a rate limiter. X-Lane can do nothing to maintain timeliness for those packets, and neither should it as the service has violated the protocol. To ensure an already broken interaction does not waste resources, the controller decreases priority of that service's packets right after the drop, voiding their timing guarantees. Then, the jitter reduction is communicated to services sharing queues with the misbehaving one, and the latter is notified by `bandwidthChanged` with `resources.priority` set to `UNBOUNDED` and `reason` to `BW_EXCEEDED`. This service may recover later with `changeBandwidth`. Further, switches' meter tables are used to identify services that behave well but *underutilize* resources. The controller reclaims a portion of their bandwidth through the `bandwidthChanged` callback with higher period and `reason` set to `BW_UNUSED`.

In the extreme scenario when a service keeps violating the protocol and/or drives its bandwidth allocation to zero by not utilizing resources, the controller terminates the protocol unilaterally with `bandwidthTerminated`.

Fine-grained jitter control with sub-lanes. Earlier, we saw newly set up tunnels adding jitter to existing ones and vice versa, whose effect we incorporated in periodic protocol parameters. Certain combinations of services require a different approach. A low-traffic jitter-sensitive service (e.g., failure detection, cf. §5.1) and a throughput-oriented one needing a "small enough" latency bound (e.g., replication, cf. §5.2), affect each other in very unequal ways leading to suboptimal overall performance. The controller addresses this issue through virtual sub-lanes — virtual controller instances that use different priority levels for timely communication, isolating services in a higher-priority sub-lane from lower-priority sub-lanes. This separation needs only be reflected at the tunnel setup, where lower-priority tunnels must include jitter from higher-priority ones but not the other way around.

4 Overcoming Jitter in Data Centers

Comprehensive mitigation of jitter sources due to interference with the rest of the data center (outlined in §2.3) is key to achieving latency with tight bounds. In what follows we describe our technique and discuss implementation details.

4.1 Bit Flips Errors in Links

Most of the messages transmitted via X-Lane are expected to be much smaller than the MTU size. To reduce the data transmission overhead, X-Lane tries to pack multiple data chunks into a single physical packet. The increased chance of packet loss due to bit flip errors is mitigated by using two custom error correction schemata that provide the same mean time to fault packet acceptance as layer 2 headers (i.e., 10^6 years with

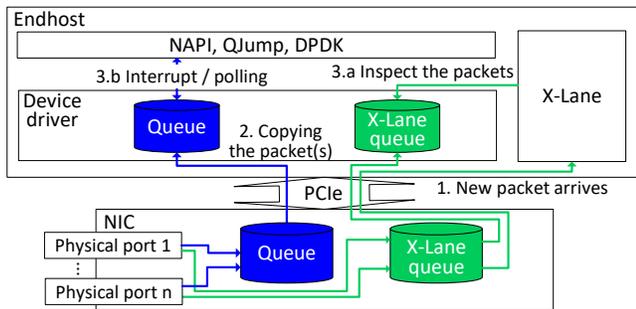


Figure 2: Overview of packet reception on commodity HW.

a bit error rate of 10^{-12}), while supporting either up to 55 chunks of 26 bytes per MTU or up to 40 chunks of 36 bytes (depending on the schema). Both schemata use a specific choice of cyclic redundancy code (CRC) polynomials.

4.2 Buffer Overflows and Jitter in Switches

Endhost NICs have a large amount of buffer memory available, allowing them to enqueue large numbers of packets before they are constrained to drop some. In contrast, commodity switching HW has a much smaller amount of (shared) buffer memory, that is commonly exceeded in the case of congestion, leading to packet losses ultimately hampering latency and jitter bounded communication. Commodity switches using an ASIC as forwarding processor can have their shared buffer split in multiple queues that are populated with packets from incoming traffic and are processed following a given scheduling strategy.

X-Lane uses a strict priority scheduler to realize the TE approach introduced in §3, to serve queues in order of priority, i.e., a non-empty queue is chosen over any other queue with lower priority. For each switch handling X-Lane’s flows, the X-Lane controller (cf. §2.2) dedicates the switch’s highest priority queues to X-Lane, and adapts the queues’ size to the expected load. X-Lane packets are therefore processed as fast as possible, reducing both jitter and the risk of packet drops since packets are processed before the queue is full. Furthermore, commodity switches are tailor-fitted to forward packets, they thus do so deterministically in the ns range [2].

4.3 Jitter in Endhost Commodity Hardware

While the standard network stack built upon endhost commodity HW can be used for throughput-oriented communication, the many sources of jitter it contains preclude X-Lane from using it for communication with bounded latency. Fig. 2 depicts how packets are handled when received on X-Lane (green) compared to the regular system (blue); X-Lane focuses on timestamping packets as early as possible to minimize stamping jitter, doing so even before their payload is inspected,

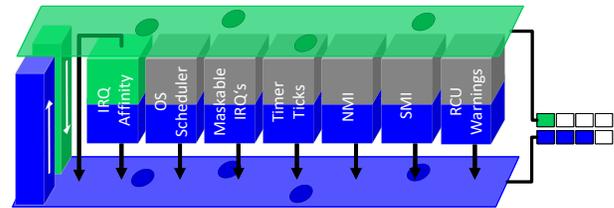


Figure 3: X-Lane is pinned to a dedicated core on which the sources of preemption (cuboids) are entirely (gray) or partly (green) disabled. The regular system is running on all other cores with all the side effects.

therefore performing *optimistic timestamping*. In the following, we give an overview of the measures implemented in X-Lane to drastically reduce jitter and latency of transmitting packets atop endhost commodity HW and SW.

First, at least a CPU core must remain available at all times for X-Lane services to promptly send and receive packets to/from other services running on other endhosts. To do so, X-Lane runs on a dedicated core (§4.3.1) and shunts preemption on it to minimize completion time of X-Lane services (§4.3.2). Second, packets must be copied between the CPU, for processing, and the NIC, for remote exchange, while avoiding jitter-prone kernel memory management (§4.3.3). Fig. 3 gives an overview of preemption sources X-Lane disables compared to a regular system.

4.3.1 Highly Responsive X-Lane Dedicated CPU Core

Execution slots on CPU cores are managed by the OS kernel scheduler which, typically, distributes these slots in a fair manner across all applications to avoid resource starvation. Timing-sensitive tasks are therefore regularly preempted to leave room for other tasks, increasing both latency and jitter for the former. Even earliest deadline first (EDF) schedulers [43] are affected by their jitter-prone environments and cannot guarantee the highest degree of responsiveness for such tasks. Furthermore, CPUs can switch between power consumption modes (i.e., C-states defined by the ACPI standard) to save energy when idle but need to wake up from an idle mode to execute a task, hampering response time [18].

X-Lane thus is *pinned* to a core, and isolates it from the scheduler to avoid task preemptions for a better response time. We call this core *X-Lane’s core* as it is (almost) exclusively managed by X-Lane. X-Lane’s core is isolated by including it in the `isolcpus` kernel boot parameter. To avoid costly wake-ups, X-Lane’s core remains in the highest active state by setting the following kernel boot parameters: `cpuidle.off=1`, `powersave=off`, `processor.max_cstate=0`.

4.3.2 X-Lane's Uninterrupted Execution

Interrupt request (IRQ) signals are generated by HW devices, e.g., I/O devices or CPU, to notify a core of an event to handle. The CPU preempts the task it is running to treat the received IRQ, which in effect increases the task's completion time and completion jitter due to the unpredictability of these IRQs.

We mitigate these delays by shielding X-Lane's core from as many IRQs as possible, as overviewed in Fig. 3. Those that cannot be ignored see their impact reduced (e.g., timer ticks).

IRQ affinity. On multi-core systems, IRQs can be distributed among cores statically — IRQs are always routed to the same core, or dynamically — IRQ affinity is set such that IRQs are handled by the core running the lowest priority task.

Most IRQs are routed away from X-Lane's core via a static distribution while other cores use a dynamic distribution, achieved by changing each IRQ's `smp_affinity` file in `/proc`.

IRQ masking. Some IRQs cannot be re-routed by setting their IRQ affinity, such as inter-processor interrupts (IPIs) that target a specific core. These IRQs can however be masked to prevent them from preempting the targeted core.

X-Lane uses the `local_irq_save(int state)` kernel function to mask IRQs before it executes an X-Lane service, and it restores the IRQ state afterwards using `local_irq_restore(int state)` with the same parameter. The masked IRQs are routed to other cores, by adapting their affinity, to preserve the correct operation of the system.

NMI watchdog. The Linux kernel integrates a watchdog timer that regularly sends non-maskable interrupts (NMIs) to each core to test for HW failures; it halts the system if the HW does not handle the NMI. There exists no standard kernel mechanisms to ignore the watchdog's NMIs.

X-Lane prevents these jitter-inducing NMIs by disabling the watchdog using the `nowatchdog` kernel boot parameter.

Timer ticks. Timer ticks are a special type of IRQs originating from CPU-local timers or external timers. They are used to run routines at a set frequency, typically between 100 and 1000 Hz, as configured in the kernel [46]. In our experiments, we have observed a substantial processing time for each of these interrupts, ranging from 1.5 μ s to 50 μ s.

X-Lane mitigates timer interrupts by configuring the kernel with the `CONFIG_NO_HZ_IDLE=y` option and adding X-Lane's core to the `nohz_full` kernel boot parameter, which sets the given core to adaptive-tick mode. While this mode does not completely oust interrupts, it greatly reduces their frequency to 1 Hz, offering significant timing improvements. For even greater improvements, X-Lane masks timer interrupts during the execution of its services. Masking these IRQs however does trigger warnings from the read, copy, update (RCU) stall detector that preempts the tasks on the masked cores.

RCU warnings. The RCU stall detector issues a warning if a core is looping (1) in an RCU read-side critical section or (2) with interrupts and preemptions disabled. The stall detector triggers these warnings, i.e., time-wise unpredictable offloadable callbacks, once its grace period is over.

The RCU stall detector issues warnings to X-Lane's core as a side-effect of masking timer (and other) interrupts on them. X-Lane thus offloads RCU callbacks to other cores by configuring the kernel with the `CONFIG_RCU_NOCB_CPU=y` option and adding X-Lane's core to the `rcu_nocbs` kernel boot parameter. Further, less callbacks are triggered and offloaded by increasing the grace period of the RCU stall detector set in the `rcu_cpu_stall_timeout` kernel boot parameter.

Unmaskable SMIs. System management interrupts (SMIs) are x86-specific unmaskable interrupts that force *all* cores to switch to system management mode to run safety-related tasks. These thus monopolize all cores for up to milliseconds, creating jitter. Some SMIs are critical to the safety of the system/HW such as the ones forcing cores throttling to prevent overheating and HW damage. These SMIs however are rare and typically do not happen in nominal scenarios.

To prevent SMIs and still protect system health, core throttling is disabled in the BIOS and X-Lane manages fans itself.

4.3.3 Packet Transfer Between X-Lane's Core and NIC

Sending and, in particular, receiving packets on an endhost is not a task as straightforward as on a switch. The complexity of this task lies within the memory management and device management modules of the Linux kernel that contain design decisions typically favoring fairness, i.e., reducing overall latencies, over prioritizing accesses for select applications.

To reduce latency and jitter, X-Lane optimizes (1) how packets are copied between X-Lane's core, that packs outgoing and unpacks incoming packets, and a NIC, that encodes/decodes packets to/from the wire, and how (2) these two devices notify one another that a packet is ready to be handled by the other.

Packet copy. When booting, the NIC's driver initializes a queue on the NIC for outgoing packets waiting to be sent (i.e., TX ring buffer), and two queues for received packets waiting to be processed by a CPU core (i.e., RX ring buffers): one on the NIC and one in the main memory. Queues hosted on the NIC are accessible by every CPU via DMA over PCIe. However, different cores experience different access timings since computer architectures nowadays have non-uniform memory accesses (NUMA). As such, both CPU and the main memory are split into several NUMA nodes; memory accesses and device accesses via PCIe within the same NUMA node are faster than across nodes as the latter are forced to use the slower QuickPath interconnect (QPI) link.

X-Lane operates its *dedicated RX ring buffers*, one on the NIC and one in the main memory (X-Lane queues in Fig. 2), for packets received on the lane to prevent jitter from the regular system packets' head-of-line blocking. The TX ring buffer remains unaffected as there is no risk of head-of-line blocking when the NIC transmits packets. In addition, X-Lane selects its dedicated core such that it runs on the NUMA node that the NIC's PCIe lanes are connected to, thus avoiding the QPI link when performing a DMA to the NIC to send or receive packets.

Packet notification. While the NIC constantly polls its local TX ring buffer, populated by cores, and thus does not need any extra step to send packets, the NIC driver running on a core must be informed by the NIC that a packet is waiting to be processed in an RX ring buffer. The driver can be notified by: (1) receiving an IRQ sent by the NIC for each received packet, which is fast but inefficient for bursty traffic that creates a lot of IRQ masks, or (2) regularly polling the NIC's RX ring buffer (as with DPDK [21]), that fetches packets in batches but incurs a latency penalty for older packets (at the front of the queue) and for low polling frequencies.

X-Lane uses the IRQ-based approach to optimize delivery timing. X-Lane's core is not subject to bursty IRQs as the bandwidth is carefully managed and smoothed by the X-Lane controller (cf. § 2.2). As shown in Fig. 2, a NIC receiving a packet sends an IRQ to X-Lane's core, set with a fitting IRQ mask, using receive flow steering [47] (step 1). In response, X-Lane's core timestamps the packet, doing it as early as possible to minimize pre-stamping jitter, and copies the packet via DMA from X-Lane's queue in the NIC to X-Lane's queue in the main memory to prepare it for inspection (step 2). X-Lane then shares the packet timestamp with the application via the X-R bridge and only delivers the unpacked payload once it has been inspected (step 3.a), also via the X-R bridge. X-Lane does not change how packets are handled on the regular system, e.g., with NAPI, DPDK (step 3.b).

4.3.4 Endhost Implementation Discussion

Additional work *in* the kernel would further improve the readiness of the implementation. For instance, X-Lane is currently limited by the granularity of some kernel boot parameters that affect all cores (e.g., disabling the NMI watchdog) and would benefit from per-core feature selection to better isolate its core. Further, most of these features are statically set at boot time, or even compile time. A dynamic configuration would help X-Lane's adaptation at runtime, reducing its endhost footprint when X-Lane is unused. Ideally, we would be able to fully isolate cores at runtime to greatly improving X-Lane's efficiency both in terms of endhost resource utilization and implementation effort.

X-Lane currently uses one core but can scale to multiple without introducing delays as long as they are in the same

NUMA node. The implementation currently focuses on Intel Xeon architecture, but AMD's EPYC has fewer NUMA nodes yet more cores, different memory management, and PCIe 4 that could improve X-Lane's performance.

4.4 Jitter in Endhost Specialized Hardware

As an alternative to endhost commodity HW, we propose an implementation of X-Lane on recent intelligent network devices (smartNICs) that completely avoid kernel-induced jitter since they are not managed by it.

Our implementation supports Netronome's smartNICs with NFP-4000 network flow processors. The NFP-4000 natively supports programs in microC, a dialect of C, and P4 [15] via a P4-to-microC transpiler. We chose microC to implement X-Lane's services on the NFP-4000-powered smartNIC as it is more expressive than P4 despite recent developments on the latter, e.g., microC can directly access packet processing, flow processing cores, internal and external memory units.

Following the NFP-4000's architecture [8], X-Lane components are running on a flow processing island that has 12 flow processors and its own memory to buffer packets. The number of flow processors used for X-Lane can be scaled on demand to match the traffic. Unlike the commodity HW implementation, here X-Lane has direct access to the packet processing pipeline and the ingress/egress buffers closest to the physical interface which greatly reduces the jitter associated to sending/receiving packets on endhosts (cf. § 4.3.3).

5 Example Services Exploiting X-Lane

We propose two services (cf. § 2.4), a failure detector (FD) service and a state machine replication (SMR) service, that exploit X-Lane to accelerate asynchronous protocols. These services are available for applications as part of the X-KM.

5.1 Failure Detector X-FD

We leverage a periodic multicast protocol (cf. § 2.1) that resides at the core of X-Lane to propose a heartbeat-based FD, X-FD, with a heartbeat period T . Unlike \mathcal{HB} [11] that outputs a vector of message counters to the application, X-FD tracks the state of remote processes in an alive table stored in the X-R bridge that can be read by any application.

X-FD operates in three successive steps. First, a user space application increments a timer value in the R-X bridge at least once per period T . Due to the jitter-prone nature of the application, the value update period must be much smaller than T (e.g., $T/3$ in § 6.4). Second, X-FD reads the corresponding value once per T from the R-X bridge and uses it for the heartbeat message, which is sent through X-Lane every period. Finally, when the destination endhost receives the packet at the queue dedicated to X-Lane on the NIC, X-FD optimistically timestamps the packet (cf. § 4.3.3) and, while the

packet’s payload is being analyzed, the alive table is updated with sender IP, port and last alive message timestamp.

5.2 State Machine Replication X-Raft

We offer a second service by adapting Raft [45], a popular SMR protocol [35, 36], to X-Lane in the form of the X-Raft service — a faster version of Raft using the periodic multicast protocol (and Raft’s acks).

We adapted the well known etcd Raft [4] without any structural modifications to the algorithm or to its different phases (e.g., leader election, log replication/recovery, membership).

X-Raft uses the R-X bridge to enable an application to interact with the SMR (e.g., to propose a value) and uses the X-R bridge to notify the application. Leader election and consensus rounds are performed in X-Lane without interacting with the application.

X-Raft uses X-FD to detect process failure and initiate leader reelection if needed. Throughput-oriented log replication packets are sent via a lower-priority sub-lane with a very small period while commit statements are piggybacked on X-FD’s low-jitter periodic messages. In addition, X-Raft batches parallel consensus instances in one packet akin to other consensus protocols [57]. Timeouts are greatly reduced thanks to X-Lane’s low latency.

The log hosted by the leader is a buffer for uncommitted inputs; an input i is removed from the log when all replicas commit to a state that includes i . X-Raft uses a ring buffer for the log that is big enough to store the logs long enough for all replicas to commit a state or fail. The commit state pointer on the ring buffer is updated when replicas commit a new state.

6 Evaluation

In this section we assess the performance of X-Lane by first evaluating the latency and jitter of the underlying switching HW (§ 6.1), followed by extensive evaluation of X-Lane’s communication timings (§ 6.2) and their variability (§ 6.3). We then evaluate the X-Lane-enabled services by measuring latency and accuracy of the FD service (§ 6.4), and latency and throughput of the SMR service both in isolation and once integrated in the Redis key-value store (§ 6.5).

Tab. 1 presents an overview of the implementation efforts behind each endhost component of X-Lane.

6.1 Hardware Setup

We ran our evaluation in a production data center of SAP SE hosting Arista 7280CR-48 [3] switches and 17 servers with Intel Xeon E5-2680 v4 at 2.40GHz (26 cores, 52 threads), 1 TB RAM, Mellanox ConnectX-4 4x10 GbE [7] and Intel XL710 4x10 GbE [6] as commodity NICs, and Netronome Agilio CX 2x10 GbE [8] smartNICs.

Table 1: Number of lines of code for each X-KM component.

Core component	#LoC	Service (cf. § 5)	#LoC
Controller client	476	X-FD	223
NIC bridge	515	X-Raft	843
SmartNIC bridge	163		

Switches’ timing impact. We evaluated the impact of switches on latency and jitter by running multiple benchmarks with varying numbers of switches between endhosts. We observed a stable latency overhead per switch of 3 μ s for unicast and 6 μ s for multicast with no measurable jitter beyond this difference, as expected [23]. We also evaluated the accumulated impact of switches in common data center topologies [5], by running benchmarks up to a 4-hop topology; it only impacted latency, not on jitter. For this reason, we evaluated X-Lane and its services on a 1-hop topology. This topology simulates in-rack computing that represents the majority of communication in optimized systems [5].

Note that the Arista 7280CR-48 switches we used are much slower than, for instance, switches from the Arista 7150 series with processing times of 350 ns according to their data sheet [2]. *Theoretically*, such switches could thus reduce the latency of our setup by at least 2.6 μ s, without affecting jitter.

6.2 Timing Observations

Most related works focus on reducing overall latency and maximizing network utilization, this work emphasizes jitter as another, crucial, dimension for many applications and in particular coordination tasks. Hence, we compare latency and jitter of three variants of X-Lane to each other, against QJump [24], and DPDK [21]. DPDK was used at a lower level by, and thus frames the performances of, many related works on low latency (e.g., Homa [44], Fastpass [48]), high performance OSs (e.g., IX [14], ZygOS [50]), and high performance SMRs (e.g., HovercRaft [33]) (cf. § 7).

Setup. We compare five configurations — DPDK, QJump, and three variants of X-Lane. The two main variants are specific to the used HW, and the third serves as a baseline:

X-Lane_{SNIC}: X-Lane on intelligent network devices;

X-Lane_{COM}: X-Lane on commodity HW;

X-Lane₀: X-Lane_{COM} with modifications made to CPU scheduling (cf. § 4.3.1) and interrupts (cf. § 4.3.2) disabled.

We report DPDK’s values using default settings as its maximum jitter did not vary measurably when varying its settings, only the number of packets with such high jitter.

We measured latency and jitter of the periodic unicast protocol on all configurations. We report latency as the time between a process sending a packet and the receiving process timestamping said packet. Sender and receiver processes

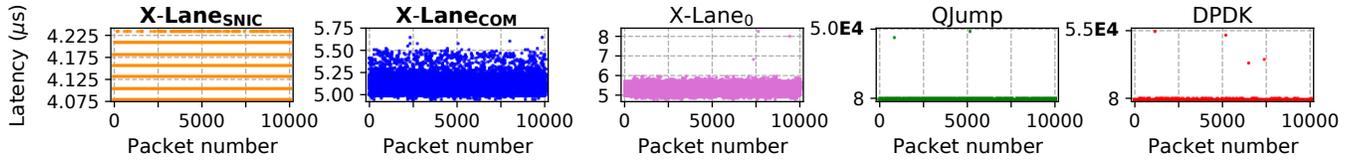


Figure 4: Overview of 10,000 packet latency (in μs) on the three X-Lane variants, QJump and DPDK. Note y-axes greatly vary.

Table 2: Summary of X-Lane’s timings showing 0th, 50th, 99th, 100th latency λ percentiles (in μs), maximum jitter δ_{max} (in ns) from λ_{min} , and a metric based on probability bound (i.i.d. assumption) for $10 \times \lambda_{99\text{th}}$ violation over next 100,000 packets. Replacing our Arista 7280CR-48 by an Arista 7150 could in theory reduce all latencies by 2.6 μs (cf. § 6.1).

Approach	λ_{min}	$\lambda_{50\text{th}}$	$\lambda_{99\text{th}}$	λ_{max}	δ_{max}	$P_{10-\lambda, 99\text{th}}^{100,000}$
X-Lane _{SNIC}	4.082	4.133	4.234	4.234	152	0.104
X-Lane _{COM}	4.938	5.130	5.446	5.649	655	0.301
X-Lane ₀	4.789	5.351	5.823	8.247	3.2E3	0.823
QJump	4.270	7.702	5.1E2	4.8E4	4.8E7	1.000
DPDK	4.103	8.904	4.0E2	5.4E4	5.4E7	1.000

are co-located on the same server to avoid cross-server clock skew; packets are still sent though the network. Processes sent packets with a 1 s period for QJump and DPDK due to high jitter, and a 10 ms period for X-Lane.

Dataset. The runs resulted in 181,440,000 packets for each approach, sampled over 21 days in a production data center of SAP SE. X-Lane variants ran with substantial cross-traffic and varying endhost utilization (up to an average CPU usage of 90%) while DPDK and QJump ran on an idle network of idle endhosts, setting the bar much higher for X-Lane. All possible point-to-point connections between servers were evaluated.

Latency and jitter results. Overall the results reveal: (1) holistic approaches (X-Lane_{SNIC}, X-Lane_{COM}) perform better than network-focused ones (X-Lane₀, QJump) and endhost-focused ones (DPDK), (2) offloading X-Lane to smartNICs (X-Lane_{SNIC}) further improves timings compared to the already efficient commodity HW approach (X-Lane_{COM}).

Tab. 2 overviews the timing measurements while Fig. 5 complements the table by exhibiting the main percentiles of the packet jitter distribution of each configuration. Even when running on commodity HW, X-Lane_{COM} shows great performance benefits compared to QJump and DPDK, e.g., $1.501 \times$ and $1.735 \times$ lower median latency, and $72,758 \times$ and $81,816 \times$ lower maximum jitter, respectively. Unsurprisingly, the results indicate that offloading X-Lane to an intelligent network device achieves the best results across the board. Compared to X-Lane_{COM}, X-Lane_{SNIC} achieves $1.241 \times$ lower median latency and $4.377 \times$ lower maximum jitter. As jitter is the most

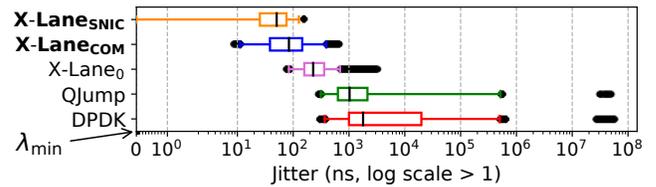


Figure 5: Distribution of X-Lane’s packet jitter δ (in ns, log scale for data > 1). A jitter of 0 corresponds to the packet(s) with minimum latency λ_{min} within a dataset. Boxes are 25th/75th percentiles, black bars are medians, whiskers are 1st/99th percentiles, further data points are grayed out.

important factor for coordination tasks in distributed systems, X-Lane shows its drastic reduction of maximum jitter makes it a prime candidate for such tasks (cf. § 6.4, § 6.5). The difference in timings between X-Lane₀ and X-Lane_{COM} shows the importance of tuning on endhost commodity HW (cf. § 4.3) to reduce maximum jitter, i.e., tail latencies.

Fig. 4 further shows the individual latency of 10,000 packets among the highest outliers. Some packets for QJump and DPDK, i.e., the “outliers” in Fig. 5, dramatically increase the jitter implying all the bad side-effects for coordination.

6.3 Latency Bound Tightness

We study the variability of the results obtained after 21 days of sampling in § 6.2 to determine the tightness of X-Lane’s bounds. We first focus on packets whose latencies are beyond the 99th percentile, then propose an extrapolation using a simple probability-based metric.

Beyond the 99th percentile. Fig. 6 depicts percentiles characteristic of tail latency based on the sampled dataset. DPDK, which has the highest λ_{avg} , makes one jump at the 99.98th percentile. At the 99.997th percentile, we see once again that as more of QJump’s “outliers” are taken into account, there is a sharp increase in tail latency. All X-Lane variants exhibit a stable behavior with X-Lane_{SNIC} being the most stable followed by X-Lane_{COM} and X-Lane₀. Another indication that X-Lane fully bounds the latency is the relative jitter defined as $(\lambda_{\text{max}} - \lambda_{\text{min}}) / \lambda_{\text{avg}}$. While the relative jitter is ≈ 0.02 for X-Lane_{SNIC}, ≈ 0.13 for X-Lane_{COM}, and ≈ 0.36 for X-Lane₀, the values for DPDK and QJump are orders of magnitude higher: $\approx 1,807$ and $\approx 1,113$, respectively.

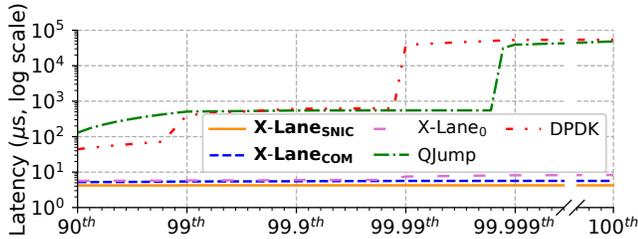


Figure 6: Tail latencies at different percentiles (different numbers of “nines”) observed over 21 days.

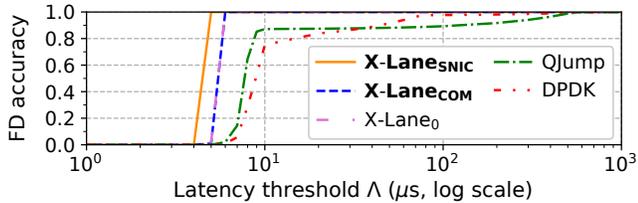


Figure 7: Accuracy of X-FD achieved when varying the latency threshold Λ . An alive process is incorrectly suspected of having failed if its heartbeat latency is greater than Λ .

Probability-based metric. We consider as a metric the probability of having among the next N packets *at least one* with latency exceeding λ , $\lambda > \lambda_{\text{avg}}$. We cannot get that probability’s true value, so we use instead an upper bound P_λ^N under a simplifying assumption that the law of large numbers applies; i.e., packet latencies are independent and identically distributed, and we have performed enough experiments for sample mean λ_{avg} and variance σ^2 to be close to their true values. We derive the probability bound P_λ^1 for a single violation from the following tail-bound: $P_\lambda^1 \leq \sigma^2 / (\lambda - \lambda_{\text{avg}})^2$. By using an independence assumption we further get $P_\lambda^N \leq 1 - (1 - P_\lambda^1)^N$. P_λ^N is a rough bound used only as a metric: the smaller its value is for an approach, the less that approach is prone to outliers.

Tab. 2 shows the probability to violate an SLO of $10 \times \lambda_{99\text{th}}$ over 100,000 packets. The results support a greater reliability of X-Lane’s measured latency over that of QJump and DPDK.

6.4 Failure Detector X-FD

We implemented the X-FD service (cf. § 5.1) atop all five configurations described in § 6.2 to compare the accuracy and completeness they provide in practice. We ran X-FD with 17 servers and a heartbeat period T of 1 ms whose value is incremented in an application every $T/3$. We varied the latency threshold Λ after which a process p is suspected of failure by others if no message was received from p in Λ .

Fig. 7 shows the rate of correct detection (i.e., accuracy) of the FDs with various threshold Λ (i.e., timeliness of completeness). We omitted T in the computation of the threshold. In practice, X-FD implemented on X-Lane reached perfect accuracy with practical thresholds well below 8 μs , and even below 5 μs for X-Lane_{SNIC}. QJump reaches $\approx 90\%$ accuracy within

10 μs but struggles for *a few milliseconds* for the remaining 10% needed for perfect accuracy. DPDK takes longer.

These results mean for instance that X-Lane can detect leader failures (e.g., in Raft [45]) orders of magnitude faster than its “low-latency” counterparts. Re-elections can promptly start hence greatly improving liveness.

6.5 State Machine Replication X-Raft

We implemented X-Raft (cf. § 5.2) using X-Lane_{COM} and evaluated it against etcd Raft [4] by measuring the latency and throughput of write requests (i.e., operations) in groups of 3 to 9 processes, one per server. The configuration was evaluated by having an application send write requests to the group. Latencies were measured as the time between the user space sender emits a request and the time it is available for all user space applications in the group. Accesses to the log, hosted in a RAM disk, were thus not included in the latencies. The sender emits once 10 M write requests whose size follows a truncated normal distribution: min = 1 B, max = 10 MB and observed mean = 25.6 B, standard deviation = 10 B.

Fig. 8 shows X-Raft performs much better than etcd both in terms of average latency, 15.7 μs for X-Raft, 26 ms for etcd, and average throughput, 96 MB/s for X-Raft, 1.1 MB/s for etcd. We note that, compared to a unicast protocol, X-Raft experiences 3 μs of added delay due to the switch processing multicast (cf. § 6.1). Unlike etcd, X-Raft batches requests before sending them and relies on multicast that scales well with regard to group size. etcd’s bandwidth requirement however is linearly proportional to group size.

Treating write requests as operations, with 25.6 B mean request size, X-Raft achieves 3.7 M ops/s mean throughput. As a comparison, HovercRaft [33] achieves 1 M ops/s with 24 B requests but uses programmable switches, and NOPaxos [40] achieves 250 k ops/s (unknown size) but centralizes traffic.

Redis integration. To evaluate the genericity of X-Lane, we replaced the default inconsistent replication protocol of the Redis key-value store [9] with X-Raft. The result, a strongly consistent replicated key-value store, only took 26 lines of code of integration. Fig. 8 shows latency and write throughput for Redis and Redis+X-Raft with 3-9 servers. X-Raft reduces latency 18 \times on average and increases throughput 1.5 \times .

7 Related Work

Distributed coordination and failure detection. Over the years, several authors have explored improvements of coordination for distributed systems but only considering individual components or specific problems. Seminal works like mostly-ordered multicast [49] and unreliable ordered multicast [40] are multicast approaches where the ordering is done at the switches. Both approaches greatly improve the

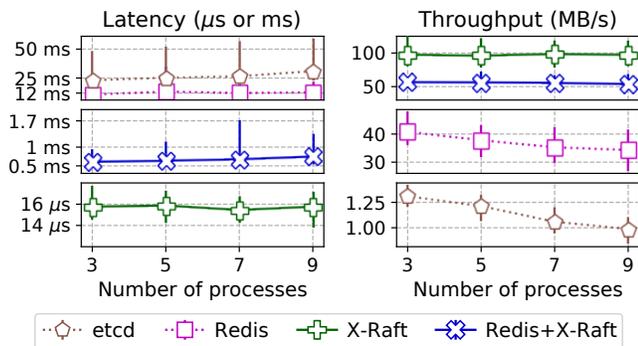


Figure 8: Write latency and throughput of X-Raft, etcd Raft, and Redis stand-alone vs with X-Raft. Mean values are plotted with min-max vertical bars.

Paxos [37] consensus protocol thanks to in-network ordering. R2P2 [34]-based HovercRaft [33], NetPaxos [17], and Consensus in a Box [29] similarly leverage switches for consensus protocols; like the Albatross [38] membership service, they do not give guarantees under an overloaded network. Their main goal is to speed up resolution of individual services via specific switch instrumentation, without considering other instances of the same protocol, other such protocols, or the network as a whole. Additionally, these approaches do not include controlled interaction to the endhosts' user space required for many jitter-bounded applications (e.g., FDs).

Silo [32] shows feasibility of guarantees without constraining network elements; the guarantees provided are however not strong enough for applications like FDs in terms of jitter and packet loss. Falcon [39] focuses on what the network needs to provide to implement a perfect (reliable) FD, rather than how it can do so, and resorts to program-controlled crashes when the FD falsely suspects processes of being crashed due to missed timeouts, contradicting reliability.

Low latency. In recent years there were numerous proposals for achieving low latency network communication. The introduced approaches typically bound latency at the 99th percentile. The reason for the 99th percentile is that it is hard to deal with the sources of jitter in a complex system (cf. §4.3). Tails of the tail [41], a seminal work in this area, identifies major jitter sources on endhosts, but does not consider the network, and focuses on 99th and 99.9th percentile latency, not 100th. Another path leading work is QJump [24] which proposes to achieve bounded latency on commodity hardware, but focuses on queues' priorities for low latency delivery and does not consider sources of jitter on endhosts (cf. §4.3).

The DPDK framework is known for its fast and efficient poll mode drivers and fast packet processing capabilities. It has a wide range of driver implementations for various NICs. The DPDK developers have restructured and implemented a majority of the network device driver code and structure.

DPDK operates by polling the network device from the user space application, which allows the programmer to harvest network packets bypassing the kernel network stack completely. As mentioned in §6.2 many works build on DPDK, e.g., Homa [44], Fastpass [48], IX [14], ZygOS [50]. These approaches try to optimize utilization and 99th percentile latency. Thus, they could be applied for the regular system but as shown in §6.3 are insufficient for X-Lane.

Time synchronization. DTP [53], Huygens [22] and Sundial [42] are time synchronization schemes for data centers with precision below 100 ns. However, time synchronization alone does not enable interactions with bounded latency.

Endhost synchrony. Efforts on achieving real-time (RT) guarantees for commodity OSs like Linux are related to X-Lane. RTLinux [51] is a real-time OS microkernel running the entire Linux OS as a fully preemptive process. RTLinux treats every process as having RT requirements, while X-Lane can treat a process in fair scheduled manner, or with even stronger RT guarantees; traditional RT schedulers, e.g. EDF [43], can actually not guarantee that a specific task is performed by a given deadline, as they can not predict the system environment and are influenced by system service executions.

8 Conclusions

X-Lane implements unprecedented low latency and jitter for coordination interaction crucial to the liveness of many applications in data centers. As this is not needed for all types of distributed interaction, X-Lane confines these bounds to an express lane, which is carefully isolated from the regular existing environment for best-effort traffic both in the network and at the endhosts. X-Lane's original design uses commodity SW and HW, and smartNICs when available.

X-Lane opens up many avenues for future research, e.g., which parts of an application best benefit from X-Lane, how to design and optimize coordination protocols accordingly. We are exploring extensions and refinements of our work such as expanding the endhost implementation (cf. §4.3), adding services (e.g., clock synchronization), and enhancing X-Lane's safety towards practical synchronous services.

Acknowledgments

We thank the anonymous reviewers and our shepherd Dan Ports for their valuable feedback. This work was partially funded by ERC Consolidator grant #617805 (LiveSoft), DFG Center #1053 (MAKI), SNSF grant #200021_192121 (FORWARD), SNSF grant #200021_197353 (BASIS), NSF grant #1618923, Hasler Foundation, and a Facebook Distributed Systems Research Award.

References

- [1] Apache HBase. <http://hbase.apache.org/>.
- [2] Arista 7150 Series. https://www.arista.com/assets/data/pdf/Datasheets/7150S_Datasheet.pdf.
- [3] Arista 7280R Series. <https://www.arista.com/assets/data/pdf/Datasheets/7280R-DataSheet.pdf>.
- [4] etcd. <https://github.com/etcd-io/etcd/>.
- [5] F16 - Facebook's topology. <https://engineering.fb.com/data-center-engineering/f16-minipack/>.
- [6] Intel XL710. <https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf>.
- [7] Mellanox ConnectX-4. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-4_VPI_Card.pdf.
- [8] Neutronome NFP-4000 network processor. https://www.neutronome.com/static/app/img/products/silicon-solutions/WP_NFP4000_T00.pdf.
- [9] Redis. <https://redis.io/>.
- [10] TiKV. <https://github.com/tikv/tikv/>.
- [11] Marcos K. Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication. In *Distributed Algorithms*, pages 126–140, 1997.
- [12] Apache. Hadoop. <https://hadoop.apache.org>.
- [13] Apache Software Foundation. Hadoop Distributed File System. <http://hadoop.apache.org/>.
- [14] 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 *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 49–65, 2014.
- [15] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 87–95, July 2014.
- [16] Manuel Bravo, Nuno Diegues, Jingna Zeng, Paolo Romano, and Luis ET Rodrigues. On the use of Clocks to Enforce Consistency in the Cloud. In *IEEE Data Eng. Bull.*, volume 38, pages 18–31, 2015.
- [17] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at Network Speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 5:1–5:7, 2015.
- [18] Shuhaizar Daud, R Badlishah Ahmad, Ong Bi Lynn, Zahereel Ishwar Abd Kareem, Latifah Munirah Kamaruddin, Phaklen Ehkan, Mohd Nazri Mohd Warip, and Rozmie Razif Othman. The Effects of CPU Load & Idle State on Embedded Processor Energy Usage. In *2nd International Conference on Electronic Design*, ICED '14, pages 30–35, 2014.
- [19] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. In *Communications of the ACM*, volume 56, pages 74–80, 2013.
- [20] Apache Foundation. Apache Accumulo. <https://accumulo.apache.org>.
- [21] Linux Foundation. DPDK: Data Plane Development Kit. <https://www.dpdk.org>.
- [22] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pages 81–94, 2018.
- [23] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proceedings of the 2011 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '11, pages 350–361, 2011.
- [24] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can JUMP them! In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI '15, pages 1–14, 2015.
- [25] 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 Symposium on Networked Systems Design and Implementation*, NSDI '11, pages 295–308, 2011.
- [26] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving High Utilization with Software-driven WAN. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 15–26, 2013.

- [27] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. B4 and After: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google’s Software-Defined WAN. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM ’18*, pages 74–87, 2018.
- [28] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, volume 8 of *USENIX ATC ’10*, pages 145–158, 2010.
- [29] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI ’16*, pages 425–438, 2016.
- [30] Patrick Jahnke, Vincent Riesop, Pierre-Louis Roman, Pavel Chuprikov, and Patrick Eugster. Live in the Express Lane (project website). <https://github.com/patrickjahnke/X-Lane>.
- [31] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined WAN. In *Proceedings of the 2013 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM ’13*, pages 3–14, 2013.
- [32] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable Message Latency in the Cloud. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 435–448, August 2015.
- [33] Marios Kogias and Edouard Bugnion. HovercRaft: Achieving Scalability and Fault-Tolerance for Microsecond-Scale Datacenter Services. In *Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys ’20*, pages 25:1–25:17, 2020.
- [34] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference, USENIX ATC ’19*, pages 863–880, 2019.
- [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. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. In *ACM Transactions on Programming Languages and Systems*, pages 254–280, April 1984.
- [37] Leslie Lamport. The Part-Time Parliament. In *ACM Transactions on Computer Systems*, volume 16, pages 133–169, May 1998.
- [38] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Taming Uncertainty in Distributed Systems with Help from the Network. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys ’15*, pages 9:1–9:16, 2015.
- [39] 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 Twenty-Third ACM Symposium on Operating Systems Principles, SOSP ’11*, pages 279–294, 2011.
- [40] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI ’16*, pages 467–483, 2016.
- [41] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC ’14*, pages 1–14, 2014.
- [42] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkupati, Prashant Chandra, and Amin Vahdat. Sundial: Fault-tolerant Clock Synchronization for Datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI ’20*, pages 1171–1186, 2020.
- [43] Chang L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. In *Journal of the ACM*, volume 20, pages 46–61, 1973.
- [44] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John K. 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, SIGCOMM ’18*, pages 221–235, 2018.

- [45] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference*, USENIX ATC '14, pages 305–319, 2014.
- [46] Linux Kernel Organization. NO_HZ: Reducing Scheduling-Clock Ticks. https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt.
- [47] Linux Kernel Organization. Scaling in the Linux Networking Stack. <https://www.kernel.org/doc/Documentation/networking/scaling.txt>.
- [48] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A Centralized "Zero-queue" Datacenter Network. In *Proceedings of the 2014 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '14, pages 307–318.
- [49] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, pages 43–57, 2015.
- [50] 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*, SOSP '17, pages 325–341, 2017.
- [51] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. The Real-time Linux Kernel: A Survey On Preempt_RT. In *ACM Computing Surveys*, volume 52, pages 1–36, February 2019.
- [52] Laura S. Sabel and Keith Marzullo. Election Vs. Consensus in Asynchronous Systems. Technical report, Cornell University, 1995.
- [53] Vishal Shrivastav, Ki Suh Lee, Han Wang, and Hakim Weatherspoon. Globally Synchronized Time via Datacenter Networks. In *IEEE/ACM Transactions on Networking*, volume 27, pages 1401–1416, 2019.
- [54] 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, pages 5:1–5:16, 2013.
- [55] Paulo Verissimo and António Casimiro. The Timely Computing Base Model and Architecture. In *IEEE Transactions on Computers*, pages 916–930, 2002.
- [56] Paulo E. Verissimo. Travelling Through Wormholes: A New Look at Distributed Systems Models. In *SIGACT News*, pages 66–81, 2006.
- [57] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 347–356, 2019.
- [58] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: a Fault-Tolerant Abstraction for In-memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, pages 15–28, 2012.

Understanding Precision Time Protocol in Today’s Wi-Fi Networks: A Measurement Study

Paizhuo Chen Zhice Yang

School of Information Science and Technology, ShanghaiTech University

Abstract

Emerging mobile applications involving distributed control and sensing call for accurate time synchronization over wireless links. This paper systematically studies the performance of Precision Time Protocol (PTP) in today’s Wi-Fi networks. We investigate both software and hardware PTP implementations. Our study uncovers the root causes of software PTP synchronization errors. We show that with fine-tuned system configurations and an online calibration procedure, software PTP can achieve reasonable accuracy with off-the-shelf Wi-Fi devices. Hardware PTP requires a PTP hardware timestamping clock not contained in Wi-Fi NICs. We propose a method to make use of the hardware TSF counter to emulate the PTP clock. Rigorous tests traversing various conditions show that both software and hardware PTP implementations can achieve a 1- μ s level of accuracy in today’s Wi-Fi networks.

1 Introduction

Emerging mobile devices are rich in sensing, actuation, and computational capabilities. A shared time basis is essential for coordinating multiple of these devices and fusing the data they collect to enable innovative applications [23, 44]. For example, if the cameras of multiple co-located smartphones are synchronized, the videos they record can be used for 3D view synthesis, which can shift the production of volumetric [20] and free viewpoint [12] videos from fixed and professional studios to flexible and amateur scenarios; If drones and robots are synchronized, a fleet of them can cooperate, *e.g.*, to deliver a large object exceeding their individual capacities.

There are many ways to wirelessly synchronize (sync) mobile devices. Some are laboratory prototypes [31, 41] while some are already used in commercial products, *e.g.*, GPS [38], cellular [46]. They differ in targeting accuracy and application scenarios. For general networked devices, a convenient solution is through network protocols. In this paper, we focus on synchronization in Wi-Fi networks, as Wi-Fi is prevalent and will continue to play a critical role in future edge networks.

There are several synchronization choices in Wi-Fi networks. IEEE 802.11 itself defines the Time Synchronization Function (TSF) [3], which syncs clients to the connected AP

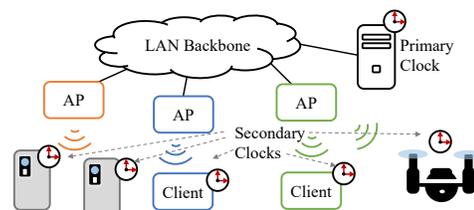


Figure 1: Scenarios of Wi-Fi Synchronization. APs are synchronized by the backbone. System clocks of Wi-Fi clients (*e.g.*, smartphones and drones) across different APs are synchronized to the primary clock. (In the context of this paper, we use Primary/Secondary clocks to replace the Master/Slave clocks used in the conventional synchronization terminology)

by resetting clients’ time to the time broadcasted in the AP’s beacons. However, TSF is only effective within a single AP’s coverage, whereas the mobile clients may span a larger geometrical area. For example, the logistics robots might operate across multiple APs to transport items. To achieve synchronization on such a scale (Figure 1), the industry’s standard practice is to use the Precision Time Protocol (PTP).

PTP is originally designed for LANs (IEEE 1588) [1] and can achieve nanosecond accuracy through timestamping the network packets. Attempts to extend PTP to wireless LAN or Wi-Fi can be traced back to the early Wi-Fi era [29]. The major difference from the wired situation is that the wireless channel introduces uncertainties in PTP timestamps [22]. This problem can be well addressed by hardware-assisted timestamping, *i.e.*, hardware PTP. The representative hardware PTP design is the latest IEEE 802.11 feature - Fine Timing Measurement (FTM) [3]. An alternative way is software PTP, which timestamps packets in the host software and hence is subject to various uncertainties caused by packet loss, channel contention, and host scheduling [22, 33]. Existing studies suggest ways to eliminate the uncertainties, *e.g.*, taking timestamps in the interrupt service routine (ISR) [33].

However, when we tried to apply the above implementations and insights to practical use, we encountered multiple friction and glitches, and finally failed. We intended to use

PTP to improve the synchronization of mobile cameras [20] for better 3D video synthesis. Our cameras are built on the Jetson Kit [14], a typical commercial off-the-shelf (COTS) development platform. For hardware PTP, while FTM seems perfect, we found that commercial FTM WNICs are mainly for positioning purposes [27] and the synchronization interface is proprietary and not publicly available. We then implemented software PTP based on existing work, but its performance is not as good as described. We found most of the related work was conducted around a decade ago, and thus many findings are no longer sufficient to explain and improve the PTP performance on current mobile CPUs and Wi-Fi NICs. To fulfill our application goal, we were forced to revisit PTP under the context of today's Wi-Fi networks and devices. We sought to uncover the factors affecting the PTP performance and ways to contain them.

A working PTP system mainly consists of two parts. The upper part, *i.e.*, the PTP daemon, such as `linuxptp` [15] and `gptp` [8], implements the PTP protocol state machine and clock tuning algorithms. The lower part timestamps PTP packets, and is finished in a way to minimize any timing uncertainties. The lower part, to our knowledge, only has proprietary implementations [11]. We develop two versions of the PTP lower part. The software PTP implementation timestamps packets in the WNIC driver. The hardware PTP implementation uses the TSF counter to timestamp packets in the WNIC hardware. They are both based on hacking and modifications on an open source WNIC driver and work seamlessly with existing PTP daemons. Their source code is available at [18]. Based on those, we conduct measurements to understand PTP performance in mobile platforms and practical Wi-Fi situations. Our key findings are:

- The major causes of software PTP's unstable performance are the power saving and performance optimization mechanisms employed in modern mobile systems. We reveal that WNIC's interrupt mitigation and CPU's idle power management are the dominating factors.
- With proper configurations, software PTP is reasonably accurate (sub- μ s bias) and stable (1- μ s jitter). A limitation of software PTP is the asymmetry interrupt responsiveness of host platforms. We propose a convenient online calibration method to address it.
- Hardware PTP can be supported by COTS WNICs by making full use of TSF counter. Although TSF counter is periodically affected by Wi-Fi TSF synchronization, our analysis suggests that it is still sufficiently accurate when the beacon frequency is as high as the default value.
- Hardware PTP is accurate, stable (sub- μ s bias error and sub- μ s jitter), and independent of software configurations. Similar to the wired PTP situation, calibration of the hardware timestamper is needed to achieve high accuracy when using different models of WNIC.

Our measurements traverse various workloads and conditions, covering both normal and extreme use scenarios. Our

analysis provides a sound and up-to-date understanding of Wi-Fi PTP performance. The implementation incorporating our findings renders an accurate and scalable synchronization solution for general mobile systems. It allows us to continue investigating 3D video synthesis from videos captured by multiple mobile cameras.

2 Background

2.1 Clocks in Computer

A clock is a tool that measures the elapse of time, and normally consists of an oscillator and a counter. The oscillator periodically generates ticks at a fixed rate of R Hz. The counter is triggered on each tick and counts the number of ticks from the time it is powered on, says T_0 . Let n denote the value of the counter, and n/R measures the elapsed time since T_0 .

In practice, T_0 is arbitrary. It is more convenient to count the elapsed time T_{off} from a chosen reference time T_{ref} . T_{off} can be measured by other clocks. At time t_i , the counter of the clock is n_i , and the "time" of the clock is defined by

$$T_i = T_{\text{off}} + \frac{n_i}{R}. \quad (1)$$

T_i measures the elapsed time since T_{ref} . In particular, when T_{ref} is a known absolute time, *e.g.*, the start of 1 January 1970, we say T_i measures the absolute time. Equation (1) maps the value of the clock counter to the time of the clock through two parameters, $\{T_{\text{off}}, R\}$. This tuple uniquely determines the time of a clock.

Modern computers contain multiple clocks that belong to different hardware modules. This paper is related to three clocks: SYSTEM clock from the host computer, TSF clock from the WNIC, and PTP clock from the Ethernet NIC. Each clock has a counter, an oscillator, and the corresponding tuple $\{T_{\text{off}}, R\}$. System clock represents the clock that the software can refer to. In our experiments, TSC clock [10] in x86 PCs and CCNT clock [6] in ARM platforms are used as the source of system clock. We use T_i^{SYS} to denote the time of the system clock, *i.e.*, system time. When describing multiple hosts, we use an intermediate superscript to differentiate, *e.g.*, $T_i^{\text{P}_{\text{SYS}}}$ and $T_i^{\text{S}_{\text{SYS}}}$ denote the system time of clock P and clock S respectively.

TSF is defined in the IEEE 802.11 standard as a mandatory feature. Every WNIC maintains a 64-bit TSF counter with a 1 MHz tick rate. TSF counter can be accessed by software through reading TSF registers. We use n_i^{TSF} to denote the counter value, and T_i^{TSF} to denote the mapped TSF time. Similarly, T_i^{PTP} denotes the time of the hardware clock of the Ethernet NIC supporting PTP protocol.

2.2 PTP Basics

PTP is defined in the IEEE 1588 standard [1] to achieve time synchronization for networked computers on the LAN scale. PTP works by timestamping PTP packets. The protocol syncing the secondary clock (S) to the primary clock (P) is explained in Figure 2. The SYNC packet is sent out

from the host of the primary clock, and timestamped by the primary clock as T_1^P . The SYNC packet is received by the host of the secondary clock, and timestamped by the secondary clock as T_2^S . In the reverse direction, the DELAY REQUEST packet triggers two timestamps T_3^S and T_4^P . FOLLOW UP and DELAY REPLY packets are used to convey T_1^P and T_4^P to the host of the secondary clock for calculating the clock offset. If the clock drift is ignored in the period of measurement, the four timestamps have the following relation: $T_1^P - T_2^S = \Delta - T_{\text{delay}}^{P \rightarrow S}$; $T_4^P - T_3^S = \Delta + T_{\text{delay}}^{S \rightarrow P}$, where T_{delay} denotes the network latency between the two hosts. In a single hop network connection like Wi-Fi, this value is stable and symmetric¹, *i.e.*, $T_{\text{delay}}^{P \rightarrow S} = T_{\text{delay}}^{S \rightarrow P}$, the offset from the primary clock can be calculated by

$$\Delta = \frac{(T_1^P - T_2^S) + (T_4^P - T_3^S)}{2}. \quad (2)$$

According to this offset value, the secondary clock can be adjusted, *e.g.*, by tuning $\{T_{\text{off}}^S, R^S\}$, to reduce any future Δ .

In practice, the synchronization accuracy of PTP depends on how precisely the four timestamps are associated with the packet transmission (Tx) and reception (Rx) events. Usually, there are errors δ between the time the timestamps are taken and the time the Tx/Rx events happen. Therefore, the actual relation of timestamps is:

$$T_1^P - T_2^S = \Delta - T_{\text{delay}}^{P \rightarrow S} + \delta_{\text{Tx}}^P - \delta_{\text{Rx}}^S \quad (3)$$

$$T_4^P - T_3^S = \Delta + T_{\text{delay}}^{S \rightarrow P} + \delta_{\text{Rx}}^P - \delta_{\text{Tx}}^S, \quad (4)$$

where $\delta_{\text{Rx/Tx}}^{P/S}$ denote the errors of Tx/Rx timestamps of the Primary/Secondary clock receptively. Equation (2) becomes:

$$\frac{(T_1^P - T_2^S) + (T_4^P - T_3^S)}{2} = \Delta - \frac{(\delta_{\text{Tx}}^P - \delta_{\text{Tx}}^S) + (\delta_{\text{Rx}}^P - \delta_{\text{Rx}}^S)}{2}. \quad (5)$$

As $\delta_{\text{Rx/Tx}}^{P/S}$ is unknown, the offset calculation by (2) contains errors and affects the synchronization accuracy.

In Ethernet hardware PTP implementations, timestamps are taken by the NIC hardware, and thus $\delta_{\text{Rx/Tx}}^{P/S}$ are relative stable and can be compensated. $|\Delta|$ calculated by (2) can be limited to sub-*ns*. Software PTP timestamps packets in software routines. The uncertainties of software stack bring about variances in $\delta_{\text{Rx/Tx}}^{P/S}$ and result in synchronization errors at the *ms*-level. In LAN situations, as hardware PTP has been widely supported by Ethernet NICs, software PTP is usually used for testing or a temporary choice when there is no compatible PTP NICs.

3 Measurement Methodology

3.1 Testbed

Hardware. The PCs are all Lenovo ThinkCentre M920t-D224 with identical hardware (Intel i5-8500 CPU, 4 GiB RAM). The motherboard has three PCIe slots. The high throughput x16 slot is directly connected to the CPU, which

¹ Assuming identical PHY rate or the transmission time is compensated.

is normally used by PCIe GPU. The remaining $\times 1$ and $\times 4$ peripheral slots are connected to the CPU through a PCIe switch in the Intel chipset. As the peripheral PCIe slots and the GPU slot have different access latency profiles [32], the WNICs in our experiments are all plugged into the $\times 1$ peripheral slot. In fact, PCIe latency is closely related to PTP performance, but it normally does not affect the accuracy, see discussions in §4.2.1 and §5.2.1.

We use the Jetson Xavier NX development kit [14] (6 ARMv8.2 cores @ 1.4GHz, 8 GiB RAM, by default, 4 cores are disabled) to evaluate the performance of mobile platforms. The Jetson board by Nvidia has a powerful GPU, which is irrelevant to synchronization, but this board is popular for prototyping video-based edge applications, hence we adopt it for the study. The board has two PCIe slots. One is used for WNIC and the other for Ethernet NIC through an M.2. to PCIe adapter [4].

We use Atheros AR9388, an ath9k-compatible WNIC, for the measurement. ath9k is a mature open source WNIC driver that we can modify and work on. In addition to WNICs, the PCs and mobile platforms are equipped with Intel I210 Ethernet NICs, which support hardware PTP and are mainly used for measuring the accuracy of Wi-Fi synchronization (see §3.3 and Figure 3).

We mainly use the above hardware platforms to present the measurement results, but our findings and conclusions are verified with several similar platforms: Hikey 970 development board [9] (8 ARM cores), ODYSSEY board [16] (Intel Celeron J4105 CPU), and ThinkCentre (Intel i5-6500 CPU).

Software. The operating system running on PCs is Debian GNU/Linux 9.8 (stretch) with kernel 4.19.37. The Jetson board uses Debian with kernel 4.9.6 maintained by the community. NONE of them are the real-time (rt) version. We use hostapd to set up Wi-Fi Access Points on PCs. linuxptp [15] is a user space daemon that implements the PTP protocol, and sends and receives PTP packets through the network socket. It is also responsible for synchronizing two clocks in a local system through the PTP kernel interface [17], *i.e.*, sync the system clock to the hardware PTP clock or vice versa. We do not modify linuxptp nor the PTP kernel interface. Instead, our implementation patches the ath9k driver to allow COTS Wi-Fi NICs to work properly with the existing PTP software stack.

3.2 Measurement Settings

When applying PTP to sync Wi-Fi connected devices belonging to different APs, Wi-Fi APs might play different roles. When they are in the transparent clock mode [1], Wi-Fi clients are directly synchronized to the primary clock, which is hosted, for example, by a computer connected to the backbone Ethernet. APs only help to forward PTP packets from/to the primary clock. However, they need to compensate for the forwarding delay, which is hard to achieve in COTS APs. To avoid such complexities, we adopt the boundary clock

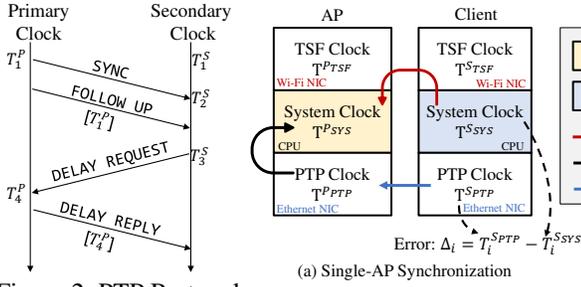


Figure 2: PTP Protocol.

mode [1], where the synchronization chain is divided into multiple domains. Wi-Fi clients are first synchronized to their primary clocks hosted by their APs. The APs are then synchronized to a primary clock connected to the LAN backbone. In this way, all of the clients, whether within the same AP or not, are synchronized.

As the LAN PTP is mature, we assume the APs are correctly synchronized. The boundary clock mode allows us to focus on the single-hop wireless synchronization within one AP. The following measurements are firstly conducted in the Single-AP situation to study the core impacting factors, and then in the Cross-AP situation to show the feasibility.

3.2.1 Single-AP Situation

The measurement settings are shown in Figure 3 (a). A PC is used as the Wi-Fi AP. It also hosts the primary clock. The associated Wi-Fi clients are either PCs or Jetson boards. The synchronization frequency (each round of procedures in Figure 2 counts as one time) is 4 times per second. For experiments other than long-term tests, synchronization data is collected for 10 mins, *i.e.*, about 2400 offset samples (the first half is dropped to wait for synchronization algorithms to converge [39]). Experiments are conducted in a typical lab environment, using the IEEE 802.11a protocol (5 GHz, single antenna) in a relatively clean channel. To evaluate the performance, we traverse the following factors:

Network and System Factors are the network protocol and system-specific factors that the end systems can control and affect.

- **CPU load.** We study the impact of computational resources of the host system by varying the CPU usage to 0%, 50%, and 100%. `stress-ng` is used to generate CPU workload at the secondary host (client) by continually performing floating point calculation. We keep the CPU load of the primary host (the AP) unchanged, as asymmetrical conditions are more likely to introduce synchronization errors. CPU turbo boost is disabled, and WNIC driver is bound to a fixed core to avoid the interference on the measurement.

- **Network traffic.** A socket program is developed to generate UDP traffic between the AP and clients. (a) The intensity of the traffic is characterized by the packet rate, *i.e.*, packet per second (PPS). The fine-grained traffic pattern is randomized by varying the inter-packet space. (b) The size of the packet is the number of bytes in the UDP payload. (c) We use `iwconfig` to fix the PHY rate from 6 Mbps to 54 Mbps.

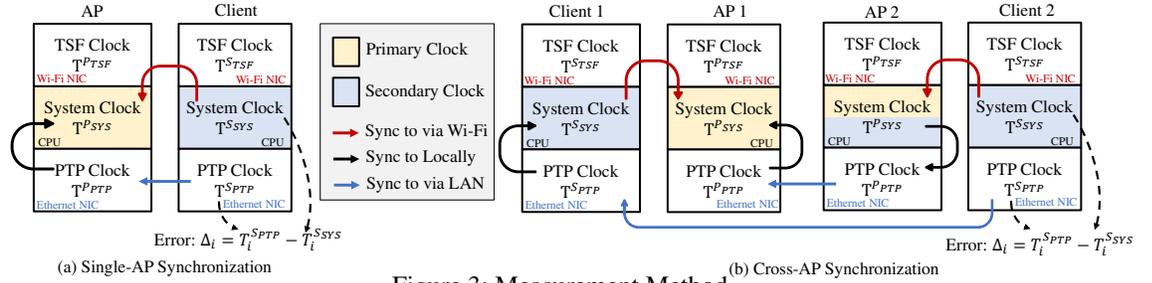


Figure 3: Measurement Method.

Note that different combinations of PPS, packet size, and PHY rate result in different amounts of *channel idle time* and *throughput*, which, according to our evaluation, have no obvious impact on the results, hence we do not list them as independent factors. Further, the network traffic is bidirectional, but we find the impact of the uplink traffic (from client to AP) is less obvious, thus only the downlink cases are presented in the paper, *i.e.*, both CPU and network load are applied to the Wi-Fi clients.

Wireless-specific Factors. Wireless networks differ dramatically from wired networks in channel properties. We consider mobility, signal strength, and neighboring interference. These factors are out of the control of the end systems and have obvious impacts on general wireless transmissions. The mobility is introduced by moving and rotating the client. The received signal strength is controlled by placing the client to different locations. The interference is generated by a signal jammer, which is another pair of PCs sending random packets with Wi-Fi carrier sense turned off.

3.2.2 Cross-AP Situation

Based on the Single-AP setup, we measure Cross-AP synchronization by syncing APs via the LAN PTP. Figure 3 (b) is the measurement setup. Client 1 and Client 2 are synchronized by first syncing Client 1/2 to AP 1/2 and then syncing AP 2 to AP 1. The two APs are directly connected by Ethernet. A scalable way to sync more APs multiple hops away is using PTP switches. Note that the system clock of AP 2 is a boundary clock serving as the primary clock for its clients, and it is also a secondary clock from the point of view of AP 1 that AP 2 is synchronized to. We denote the synchronization chain with arrows: Client 1 $\xrightarrow{\text{sync}}$ AP 1 $\xleftarrow{\text{sync}}$ AP 2 $\xleftarrow{\text{sync}}$ Client 2. In cross-AP measurements, the CPU and network load are only applied to Client 1. APs and Client 2 are load free.

3.3 Performance Metrics

In the Single-AP situation in Figure 3 (a), T^{SSYS} is synchronized to T^{PSYS} via Wi-Fi. To measure its accuracy, we use LAN PTP as the reference. The AP and client hosts are connected by an Ethernet link. The PTP clock of the client's Ethernet NIC is synchronized to AP's system clock through hardware LAN PTP. Note that the difference between T^{SPTP} and T^{PSYS} is at the ns-level, thus we use $\Delta_t = T_i^{SPTP} - T_i^{SSYS}$ to measure the synchronization error. The mean and standard

deviation (std) of Δ_i , indicating bias and stability (jitter) respectively, characterize the synchronization performance. In the Cross-AP situation, we care about how clients belonging to different APs are synchronized, and thus use the Ethernet link to connect the two clients for comparing the difference of their system clocks through the LAN PTP. The method is similar to the Single-AP situation and shown in Figure 3 (b).

4 Software PTP

This section first describes our implementation of software PTP (§4.1). We analyze its performance from a system perspective (§4.2.1). Based on that, we propose system tuning techniques to improve the performance (§4.2.2) and a calibration procedure for using with diverse host platforms (§4.2.3). We conclude the trade-offs and limitations in §4.3.

4.1 Implementation - Software PTP

The PTP mechanism does not discriminate between Wi-Fi and Ethernet, but software PTP implementations do require NIC driver support to achieve reasonable accuracy. A user space program can take timestamps on its network socket but this just leads to a trivial NTP implementation, which has accuracy at the level of tens of *ms* [37]. According to equation (3)(4), software timestamps should be taken as close as possible to the Rx/Tx events to avoid software uncertainties. Therefore, Ethernet software PTP implementations normally take timestamps through the assistance of the NIC driver. However, we have not noticed any WNIC drivers provide such support, so we implement one for ath9k.

As WNICs use interrupts to inform the operating system about the Rx/Tx events, the system clock of the host is used to timestamp the PTP Tx/Rx events in the WNIC's interrupt service routine (ISR). The timestamps are first filled into a PTP-reserved field of packets' socket buffer, and then conveyed to and used by the kernel PTP subsystem and userspace PTP applications, *e.g.*, `linuxPTP`.

4.2 Findings and Analysis - Software PTP

Through measuring our software PTP implementation, the preliminary finding is that its synchronization is neither accurate nor stable. For example, when syncing two PCs under 48 Mbps PHY, 64 Byte size, 6000 PPS background traffic, the mean error is -1134 μs with std 992.5 μs . Results of using Jetson as the client are even worse, the mean error is -1892 μs with std 2947 μs . The performance is not consistent with the existing studies having similar implementations but with older platforms [33]. Our newer platforms perform much worse. To uncover the reasons, we look into the cause of the inaccuracy.

4.2.1 Root Cause of Inaccuracy

PTP accuracy is determined by T_{delay} and $\delta_{\text{Rx/Tx}}^{P/S}$ in equation (3)(4). For transmissions between Wi-Fi clients and AP, $T_{\text{delay}}^{P \rightarrow S} = T_{\text{delay}}^{S \rightarrow P}$, so the inaccuracy is caused by $\delta_{\text{Rx/Tx}}^{P/S}$. To characterize $\delta_{\text{Rx/Tx}}^{P/S}$, we measure its value of our platforms.

As shown in Figure 4, the measurement is aided by the TSF counter. When a packet is received, a WNIC takes several actions. First, it records the value of its TSF counter as $n_{\text{Rx}}^{\text{TSF}}$ at the time when the last bit of the packet is decoded. Second, the WNIC generates an interrupt to notify the operating system and also provides a brief description of the received packet, which contains $n_{\text{Rx}}^{\text{TSF}}$. Third, in the ISR of the WNIC driver, the operating system handles the Rx interrupt and takes software timestamp T^{SYS} . Once the software timestamp is taken, we read and record the TSF counter as $\hat{n}_{\text{Rx}}^{\text{TSF}}$. Since the I/O latency of reading TSF counter is relatively stable and small, $\hat{n}_{\text{Rx}}^{\text{TSF}} - n_{\text{Rx}}^{\text{TSF}}$ is just the number of 1 MHz ticks that $\delta_{\text{Rx}} + \frac{T_+^{\text{TSF}} - T^{\text{TSF}}}{2}$ takes². δ_{Rx} can then be obtained. As when a packet is transmitted, the WNIC also reports the Tx status through an interrupt, the process of measuring δ_{Tx} is similar. Moreover, as δ_{Tx} is close to δ_{Rx} and their trend is identical, we use $\delta_{\text{Rx/Tx}}$ to describe them interchangeably.

The results of $\delta_{\text{Rx/Tx}}$ of software timestamps are shown in Table 1, from which we identify two system optimization mechanisms, Interrupt Mitigation/Throttling and CPU Power Management, which increase $\delta_{\text{Rx/Tx}}$ and cause large jitters, corrupting the synchronization performance.

Interrupt Mitigation. Interrupt mitigation is a mechanism to reduce CPU overhead by aggregating NIC interrupts [13]. When multiple NIC events occur in a period, only one interrupt is generated to notify the host system. The trade-off is, interrupts of most packets are delayed and the host system is less timely in response to network events. Interestingly, while this feature is well-known in Ethernet NICs, to our knowledge, it is rarely mentioned and evaluated under the COTS Wi-Fi context. Actually, we have not noticed any WNIC driver implements and provides the configuration interface, before our software PTP implementation.

When PPS is low (row 1 of Table 1), the mean and peak of $\delta_{\text{Rx/Tx}}$ are about 500 to 700 μs , indicating most packets need to wait for at least 500 μs for an aggregated interrupt. This is because, by default, the WNIC needs to wait for 500 μs to see if there is a consecutive packet following. When PPS increases, the inter packet space is less than 500 μs . The 500 μs threshold would lead to an unbounded waiting time, so the WNIC sets another threshold to limit the maximum waiting time, which is around 2000 μs (see peaks of row 3 of Table 1).

In cases with large asymmetric PPS, interrupt mitigation affects $\delta_{\text{Tx}}^P - \delta_{\text{Tx}}^S$ and $\delta_{\text{Rx}}^P - \delta_{\text{Rx}}^S$ in (5) by more than 2000 μs , causing the sync error at a similar level. Interrupt mitigation can be disabled by hacking the WNIC (see §4.3). After doing so, the distribution of $\delta_{\text{Rx/Tx}}$ becomes much sharper (see the peak and mean difference of row 4,5,6 of Table 1), but when the CPU load is light, its mean is still as large as 176 μs (row 4 of Table 1). The reason is the following factor.

² $T_+^{\text{SYS}} - T^{\text{SYS}}$ is the PCIe WNIC read latency. For our PC's $\times 1$ slot, it is about 1.7 μs . For Jetson's $\times 4$ slot, it is about 1.4 μs .

CPU C-state	Network		CPU Load					
	Rx IMT	PPS	0%		50%		100%	
PC C0-C8	Enabled	100	674.3	699.2	600.3	519.2	519	518.2
		1000	558.8	557.2	546.8	519.2	520.1	518.2
		5000	1093	2046	1098	2042	1088	2010
	Disabled	100	175.6	203.2	105.3	23.24	20.5	20.24
		1000	60.31	59.24	42.36	22.24	20.15	20.24
		5000	37.53	21.24	23.92	21.24	19.95	20.24
PC C0 Only	Disabled	100	19.92	20.24	19.89	20.24	20.16	20.24
		1000	19.75	20.24	19.93	20.24	19.91	20.24
		5000	19.66	19.24	19.67	19.24	19.93	20.24
Jetson C0 Only	Disabled	100	16.75	15.58	16.24	15.58	16.5	15.58
		1000	16.51	15.58	16.17	14.58	16.54	15.58
		5000	15.89	14.58	16.61	15.58	16.65	15.58

Table 1: Error of Software Timestamp. Mean error and peak (mode) of the error distribution in unit of μs . The width of the color bars in each table cell illustrates the relative size of the underlying values. The bar of mean (blue)/peak (green) logarithmically grows from right to left/from left to right. Rx IMT is short for Rx Interrupt Mitigation.

CPU Idle Power Management. Modern CPUs leverage several mechanisms to dynamically conserve power [5]. When the CPU is idle, it might choose to turn off different numbers of hardware components, represented by different CPU idle states or C-states [43], e.g., from C0 to C8 in Intel i5 8500. CPU in deeper C-state consumes less power, but takes more time to wake up to the active state C0, e.g., to handle interrupts. As a result, power management mechanisms tend to let CPUs stay in lighter C-states when there likely are more intensive tasks or more frequent interrupts. This explains why $\delta_{Rx/Tx}$ decreases when CPU load and PPS increase (row 4,5,6 of Table 1). Different CPU and PPS loads lead to 150 μs difference in $\delta_{Rx/Tx}$, causing 150 μs mean error in synchronization. CPUs can be forced to stay in C0 state via its `sysfs`, in which it takes the least latency to respond to interrupts.

4.2.2 Software PTP with Proper System Configurations

Guided by the above analysis, we improve our software PTP implementation by disabling interrupt mitigation and forcing CPUs in C0 state. When the hardware of AP and client hosts is identical, the synchronization error in most cases is within 1 μs , with std less than 1 μs (A,C,E column of Table 2). However, it becomes less accurate when the hosts hardware is heterogeneous (the right subtable of Table 2). The reasons are rooted in the following factors.

In column AA, CC, and EE of Table 2, there is a constant bias of 4.x μs . This bias can be explained by comparing row 7-9 with row 10-12 of Table 1. Compared with Jetson, PC's software timestamps are delayed by 4.x μs , causing 4.x μs residues in $\delta_{Tx}^P - \delta_{Tx}^S$ and $\delta_{Rx}^P - \delta_{Rx}^S$ in (5), which leads to the bias error. What causes the difference between $\delta_{Rx/Tx}^P$ and $\delta_{Rx/Tx}^S$? Is ARM handling WLAN interrupts more timely than x86? These questions are out of our scope, but we have the following experience. Note that as both PC and Jetson are in C0 state in row 7-12 of Table 1, $\delta_{Rx/Tx}$ does not contain too much latency to wake up the CPU. Hence, $\delta_{Rx/Tx}$ mainly consists of two parts. One is attributed to the *WNIC interrupt latency*.

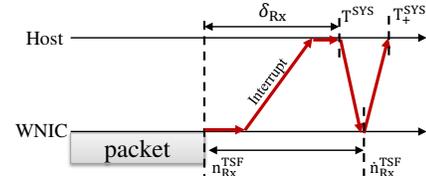


Figure 4: Using TSF Counter to Measure the Error of Software Timestamp.

There is an undocumented but stable delay between the end of Rx/Txing a packet and the issuing of the interrupt. This part is slightly different for the Rx and Tx chain, and different WNIC models. The remaining part (5 to 10 μs) characterizes the *responsiveness of the host system* in handling PCIe interrupts. Through extensive tests, we find this part varies with different host hardware, kernel versions, PCIe slots, and even CPU loads. Asymmetric interrupt responsiveness is common in host systems, which affects the PTP accuracy at the level of (less than) 10 μs .

4.2.3 Software PTP with Calibration

We discuss the calibration of the two types of bias errors.

Asymmetric Host Interrupt Responsiveness. Our insight is from the measurement method of Table 1. We note that the interrupt responsiveness is relatively stable when CPUs are in C0 and interrupt mitigation is off. Under such configurations, both the AP and client hosts can measure $\delta_{Rx/Tx}$ online when there is WLAN traffic, thus it is practically convenient. When using identical WNIC models, the *WNIC interrupt latencies* cancel out in (5), hence the secondary clock can subtract $\delta_{Tx}^P - \delta_{Tx}^S$ and $\delta_{Rx}^P - \delta_{Rx}^S$ from (5) to compensate for the bias of the host interrupt responsiveness.

Specifically, we measure all rows from 7 to 12 of Table 1. As the values are very close, only a subset is also enough. Note that the distribution of $\delta_{Rx/Tx}$ is single side and positive. The peak or mode of the measured $\delta_{Rx/Tx}$ is chosen as the specific value of $\delta_{Rx/Tx}$ for calibration (the mean value is affected by the long tail). We use the average of the peak values in Table 1 to compensate for the bias in Table 2. The compensated results are shown in Table 3. The mean error is less than 1 μs , which is comparable to the case using identical hardware (left subtable of Table 2). The std is $\times 3$ to $\times 5$ larger. The reason is the Jetson (ARM) platform is not as stable as the PC platform in handling interrupts. Jetson's $\delta_{Rx/Tx}$ contains a longer tail, which also can be observed from Table 1. i.e., PC's mean and peak is closer than Jetson's.

To investigate the long-term stability, we log the synchronization error (after calibration) for 14 hours for the same hardware settings as Table 3. A script is used to generate random CPU and network load to the client during the measurement to emulate practical scenarios. The overall mean error is 0.17 μs with std around 1.8 μs , indicating stable and accurate synchronization between the two hosts. Due to space limitations, we omit the measurement results of cross-AP and wireless-specific factors for software PTP. As the findings and conclusions are identical to the hardware PTP, the reader can

Network			PC (Client) $\xrightarrow{\text{sync to}}$ PC (AP)					
			CPU Load					
Rate	Size	PPS	0%		50%		100%	
/	/	0	0.11	0.38	-0.13	0.51	-0.19	0.48
6M	500B	100	0.13	0.37	0.17	0.5	0.25	0.54
		1000	-0.07	0.56	0.1	0.32	0.27	0.34
48M	64B	1000	0.54	0.43	-0.26	0.38	0.11	0.4
		5000	0.2	0.47	0.27	0.35	0.09	0.39
			A	B	C	D	E	F

Network			Jetson (Client) $\xrightarrow{\text{sync to}}$ PC (AP)					
			CPU Load					
Rate	Size	PPS	0%		50%		100%	
/	/	0	4.69	1.73	4.2	1.72	4.65	1.61
6M	500B	100	4.39	1.69	4.93	1.7	4.69	1.52
		1000	4.52	1.8	4.26	1.49	4.63	1.78
48M	64B	1000	4.47	1.76	4.57	1.76	4.5	1.62
		5000	4.24	1.71	4.26	1.67	4.71	1.53
			AA	BB	CC	DD	EE	FF

Table 2: Single-AP Software PTP Results. Rate: PHY rate of the Wi-Fi link (Mbps). Size: UDP payload length (Byte). PPS: packet per second. Values in the table are mean and standard deviation (std) of the synchronization error in unit of μs . The width of the color bars in each table cell illustrates the relative size of the underlying values. The bar of mean (blue)/std(red) linearly grows from right to left/from left to right. Negative mean values are shown in gray.

Network			Jetson (Client) $\xrightarrow{\text{sync to}}$ PC (AP)					
			CPU Load					
Rate	Size	PPS	0%		50%		100%	
/	/	0	0.20	1.73	-0.29	1.72	0.15	1.61
6M	500B	100	-0.10	1.69	0.43	1.7	0.20	1.52
		1000	0.03	1.8	-0.23	1.49	0.14	1.78
48M	64B	1000	-0.03	1.76	0.07	1.76	0.01	1.62
		5000	-0.26	1.71	-0.23	1.67	0.21	1.53
			AA	BB	CC	DD	EE	FF

Table 3: Single-AP Software PTP Results with Calibration. Notations are same as Table 2. This table is calculated from the right subtable of Table 2, taking the interrupt responsiveness of different platforms into account.

refer to §5.2.2 and §5.2.3.

The above calibration procedure is based on measuring $\delta_{\text{Tx/Rx}}$, which is the time between the WNIC taking TSF timestamp ($n_{\text{Tx/Rx}}^{\text{TSF}}$) and the host system taking software timestamp (T^{SYS}). When we treat $\delta_{\text{Tx/Rx}}$ as the error of software timestamp, there is an implicit assumption that the TSF timestamp $n_{\text{Tx/Rx}}^{\text{TSF}}$ and the packet event have a known and fixed association, e.g., it is taken at the end of the packet, which, however, is not be true for different models of WNICs. Some models take TSF timestamps shortly after the end of the receiving, while others might take at the middle. We term this model-dependent factor as *TSF offset*. The *TSF offset* combined with the *WNIC interrupt latency* contributes about 10 μs to latencies in Table 1. While the *TSF offset* is unknown, it is fixed for the same model, hence it also automatically cancels out in equation (5) when using WNICs of the same model. However, when using different WNIC models, it needs offline calibration. We will revisit this issue later in §5.2.1.

WNIC Interrupt Latency is another hardware-related bias source. We note that the *TSF offset* only affects the above calibration, but *WNIC Interrupt Latency* is intrinsic in software PTP. When interrupt mitigation is disabled, the interrupt should be issued once the Tx/Rx event is finished, but in practice, there is a model-dependent delay, which can be observed when using different WNIC models on identical host platforms. Due to the coupling with the *TSF offset*, its value cannot be measured with the method in Figure 4. As a model-dependent factor, it needs offline calibration similar to *TSF offset*.

4.3 Discussion - Software PTP

Software PTP timestamps packets in ISR, which is a standard software routine in the network stack. Under certain configurations, software PTP can achieve inspiring results. The accuracy is μs -level and the std is within 2 μs , which is accurate enough for many time-sensitive applications. Next, we discuss the trade-offs and limitations.

Power Efficiency. Interrupt mitigation and CPU idle states are critical for the software PTP performance. However, completely disabling them is a concern for most mobile devices, as they are limited in battery life. A general workaround is duty cycling. Since PTP does not need to frequently exchange packets, PTP packets can be pre-scheduled at given time slots, during which the mobile host switches to the “active” mode (with C0 and IMT off) for precisely timestamping PTP packets. Additionally, there are other specific alternatives. For example for retaining idle states, the AP can choose to pre-heat the client by sending a dummy packet before every PTP packet. The client will be wakened up by the pre-heat packet. For reducing the number of interrupts while keeping responsive to PTP packets, PTP traffic can be assigned to the Rx priority queue (see the following paragraph). Moreover, since the PTP measurement is periodical, it has a negative impact on the IEEE 802.11 power-saving (PS) mode. We will discuss this issue later with hardware PTP in §5.3.

Hardware Compatibility. While software PTP is mainly finished in software routines, it depends on disabling interrupt mitigation to improve the PTP performance, which is not a typical configurable parameter in COTS WNICs. We achieve this in two ways. The first is to completely disable this feature through specific registers, which might lead to too many interrupts overwhelming the host system. Another way is to prioritize PTP packets. By default, Tx interrupt mitigation is disabled by the driver. The WNIC we tested implements two hardware Rx queues with different priorities. Packets with high IEEE 802.11 QoS priority are handled by the high-priority Rx queue, and their interrupts are delivered instantly without the mitigation delay. The above two approaches are all hardware-dependent features. While we have not confirmed the compatibility of other WNICs except for the ath9k series, we think such features are functionally reasonable and are likely supported by other modern WNICs.

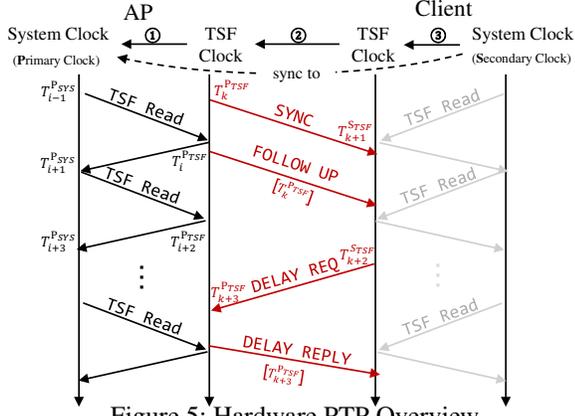


Figure 5: Hardware PTP Overview

5 Hardware PTP

This section first describes our implementation of hardware PTP for Wi-Fi devices (§5.1). Then we analyze its performance by considering the host system and hardware factors (§5.2.1). As a wireless synchronization approach, the impact of mobility and wireless channel conditions are considered in §5.2.2. Additionally, §5.2.3 demonstrates the feasibility of cross-AP synchronization. We summarize the benefit and limitations of hardware PTP in §5.3.

5.1 Implementation - Hardware PTP

Hardware PTP needs hardware counters to timestamp network packets, but how can the WNICs support a hardware feature that they are not originally designed for?

Our key insight is to treat the TSF clock as the hardware timestamping clock. The feasibility is based on two features. First, WNIC uses the TSF counter to timestamp network packets, for the purpose of protocol debugging, packet sniffing, *etc.* Second, the TSF counter can be accessed by the host operating system, through reading the TSF counter registers. The above features are the necessary conditions for a PTP hardware clock. By using the TSF counter, hardware PTP can be realized in Wi-Fi NICs in the same way as the Ethernet NICs. This subsection highlights the major procedures (①-③) shown in Figure 5:

① Generate $T^{P_{TSF}}$ and $T^{S_{TSF}}$. According to equation (1), we maintain a TSF clock for each WNIC based on its TSF counter. The time of the clock is denoted as T^{TSF} , which is determined by TSF counter and the clock parameters $\{T_{off}^{TSF}, R^{TSF}\}$.

② Sync $T^{P_{TSF}}$ to $T^{P_{SYS}}$. The host of the primary clock, *i.e.*, the AP, synchronizes its TSF clock to its system clock. The method is to read each clock and tune TSF clock's parameters $\{T_{off}^{P_{TSF}}, R^{P_{TSF}}\}$ to minimize the difference between $T^{P_{SYS}}$ and $T^{P_{TSF}}$. In practice, the latency and jitter of obtaining timestamps must be taken into consideration. We adopt a sandwich-like reading sequence to access $T^{P_{SYS}}$ and $T^{P_{TSF}}$ multiple times, and only the one with the smallest read latency ($T_{i+1}^{P_{SYS}} - T_{i-1}^{P_{SYS}}$), say k , are used to estimate their clock offset, *i.e.*, $\Delta = T_k^{P_{TSF}} - \left(\frac{T_{k-1}^{P_{SYS}} + T_{k+1}^{P_{SYS}}}{2}\right)$.

③ Sync $T^{S_{TSF}}$ to $T^{P_{TSF}}$. The host of secondary clock synchronizes its TSF clock to $T^{S_{TSF}}$ according to PTP protocol in Figure 2. Note that the WNIC hardware automatically timestamps packets with TSF counter. The value of the counter is transformed to TSF time according to ①. After each round of measurement, the time offset of two TSF clocks is estimated by $((T_1^{P_{TSF}} - T_2^{S_{TSF}}) + (T_4^{P_{TSF}} - T_3^{S_{TSF}}))/2$, according to which the host of $T^{S_{TSF}}$ tunes its TSF clock's parameters $\{T_{off}^{S_{TSF}}, R^{S_{TSF}}\}$ to minimize the offset between $T^{S_{TSF}}$ and $T^{P_{TSF}}$. Since $T^{P_{TSF}}$ has been synchronized to the primary clock $T^{P_{SYS}}$ in ①, $T^{S_{TSF}}$ is thus synchronized to $T^{P_{SYS}}$.

④ Sync $T^{S_{SYS}}$ to $T^{S_{TSF}}$. The host of secondary clock synchronizes $T^{S_{SYS}}$ to $T^{S_{TSF}}$ by tuning parameters $\{T_{off}^{S_{SYS}}, R^{S_{SYS}}\}$ like step ①. As $T^{S_{TSF}}$ has been synchronized to $T^{P_{SYS}}$ in step ②, the system clocks of two hosts, *i.e.*, $T^{P_{SYS}}$ and $T^{S_{SYS}}$, are then synchronized.

The above procedures mimic the implementation of hardware PTP of Ethernet NICs. However, unlike the Ethernet case, the TSF counter is not a free-running counter in Wi-Fi NICs. We analyze the impact of this difference in the following subsection.

5.2 Findings and Analysis - Hardware PTP

The measurement experiments are identical to the software PTP situation. The results are shown in Table 4. In all the cases, the mean error is less than $1 \mu s$ with std less than $1 \mu s$. Compared with software PTP, no obvious difference is observed when using different host platforms. This is because the timestamps are taken by the WNIC hardware, and involve no operations of the host system. A similar long-term measurement is conducted using Jetson as the client. The overall mean error is $-0.16 \mu s$ with std around $0.5 \mu s$. The sync std/jitter is smaller than the software PTP.

5.2.1 Potential Cause of Inaccuracy

While the accuracy is close to the resolution of the TSF counter (1 MHz), we investigate the following factors to provide a sound understanding on why and under what conditions the hardware PTP implementation works well.

Symmetry of PCIe Single-way Latency. In step ① and ③ in §5.1 and Figure 5. The TSF clock is obtained by accessing the TSF counter. This procedure relies on a PCIe read action to the WNIC registers, *i.e.*, memory-mapped I/O (MMIO) read. To account for the read latency, the read action is sandwiched by two timestamps of the system clock and the average of the two system timestamps is used to approximate the actual time of accessing the TSF counter. In fact, the read action contains a read request and a read reply, *i.e.*, a round trip. As a consequence, the estimation method assumes that the latencies of the read request and reply are equal, *i.e.*, the PCIe single-way latency should be symmetrical.

We are not able to quantitatively validate the symmetry due to the lack of a PCIe analyzer. However, in general, it should approximately hold, as the same problem exists in Ethernet

Network			PC (Client) $\xrightarrow{\text{sync to}}$ PC (AP)							
Rate	Size	PPS	CPU Load							
/	/	0	50%			100%				
6M	500B	100	0.18	0.37	0.2	0.44	-0.17	0.42	1	
		1000	0.14	0.41	-0.03	0.37	-0.55	0.39		2
48M	64B	1000	0.34	0.42	-0.33	0.43	0.49	0.45	3	
		5000	-0.17	0.45	0.16	0.36	0.41	0.44		4
		5000	0.24	0.34	-0.22	0.42	0.16	0.38		
		5000	0.11	0.39	0.05	0.37	-0.13	0.43	6	
			A	B	C	D	E	F		

Table 4: Single-AP Hardware PTP Results (μs). Notations are same as Table 2.

Config		PC (ROLE) $\xrightarrow{\text{sync to}}$ PC (AP)						
ROLE	BI	CPU Load						
		0%		50%		100%		
Client	1000	0.19	0.81	0.13	0.79	-0.15	0.77	1
AP	100	0.21	0.42	0.03	0.44	-0.13	0.45	2
	100	0.17	0.39	-0.15	0.45	0.03	0.38	3
		A	B	C	D	E	F	

Table 5: Hardware PTP with Free-running TSF Counter (μs). Notations are same as Table 2. ROLE is the Wi-Fi mode of host (either Client or Access Point). The TSF counter of client mode is not free-running and used for comparison. BI is short for the Beacon Interval in unit of Time Unit (TU, 1024 μs in IEEE 802.11). The default BI is 100 TU, *i.e.*, 102.4 ms.

NICs (according to results in Table 4, the impact of this factor is not obvious in our WNICs). During the test, we also notice that the read latency slightly varies with the platform load, this is again related to the power management mechanisms, but it does not break the symmetry.

We note that the read latency of different hosts or PCIe slots is usually different (recall footnote 2 in §4.2.1), but the difference does not affect hardware PTP. This is because PCIe read actions only happen locally on the host system to relate the system clock to the hardware NIC clock. Whether $(T_{k-1}^{P_{\text{SYS}}} + T_{k+1}^{P_{\text{SYS}}})/2$ can accurately estimate $T_k^{P_{\text{TSF}}}$ or not only depends on the symmetry of single-way latencies, rather than the total read latency $T_{k+1}^{P_{\text{SYS}}} - T_{k-1}^{P_{\text{SYS}}}$.

Impact of TSF Synchronization. Note that in PTP, the timestamping clock should be a free-running clock. This is only true for the Wi-Fi AP. The TSF counters of the Wi-Fi clients are automatically set to the value of the AP’s TSF counter contained in the beacon packet broadcast by the AP every 102.4 ms. As many Wi-Fi timing functions, such as power save wake up, transmission scheduling, *etc.*, rely on TSF, therefore disabling the TSF synchronization would lead to a mess in the Wi-Fi network. Therefore, we design a new experiment to compare the performance of hardware PTP using/not using free-running TSF counters.

We use two PCs as APs and set their system clocks as the PTP primary clock and secondary clock respectively. The two APs work in the same channel, which allows them to overhear the broadcast packets from each other. The unicast addresses of PTP packets are modified to the broadcast address, so that two APs can exchange PTP packets to allow hardware PTP to operate in the same way as before. Since they are all in the AP mode, their WNICs’ TSF counters are free-running. The synchronization results are shown in Table 5.

Comparing row 3 of Table 5 and row 1 of Table 4, there is no obvious performance increase or reduction when using a free-running TSF counter. However, when the beacon interval is increased, *i.e.*, the TSF synchronization frequency is reduced. The case using the free-running counter stays the same (row 2 of Table 5), while the normal configuration becomes more unstable (row 1 of Table 5). This is because the TSF counter of the Wi-Fi AP is free-running, but the TSF counter of the Wi-Fi client is periodically set to beacons. Due to the frequency drift between the two TSF counters, the value of the client’s counter jumps once it is reset by the beacons. The longer the client’s TSF counter does not sync to the beacon, the larger the jumps are. Therefore, lower sync frequency results in larger jitters in the client’s PTP clock $T^{S_{\text{TSF}}}$, and hence the sync results.

Comparing row 1 of Table 5 (BI=1000) and row 1 of Table 4 (BI=100), when the frequency of beacons is as high as the default (BI=100), the impact of TSF synchronization is limited due to two reasons. First, by default, the TSF counter is actually so frequently updated, that the jump values of the TSF counter are very small. For our WNICs, the clock drift is 20 ppm, meaning that the counter at most drifts 2 μs in each 100 TU. Second, the jumps of the TSF counter do not affect the TSF clock at the same level. This is because when an offset of TSF clocks is observed by the PTP protocol (② of §5.1), $T^{S_{\text{TSF}}}$ is adjusted by the rate part, *i.e.*, parameter R in equation (1), and several control filters are used to smooth the impact of noise and outliers. Due to the above reasons, when the client’s TSF counter is frequently updated to the AP’s, it can be viewed as a “free-running” counter, which has exactly the same rate as the AP’s TSF counter but with more jitters.

Impact of TSF Offset. Hardware PTP uses the TSF counter to timestamp packets, hence $\delta_{\text{Rx/Tx}}$ in (5) is free of the impact of the host system but is affected by the WNIC *TSF offset*. When WNICs have different *TSF offsets*, $\delta_{\text{Rx/Tx}}$ in equation (5) will not cancel out, which results in bias error. In another experiment not presented here, we replaced the client’s WNIC with another model of Atheros NIC, then a constant positive 1 μs mean error occurred in all the cases. This implies $\delta_{\text{Rx/Tx}}$ does exist and differs across models. Other indirect evidence is from the driver source code. There are registers with magic numbers for compensating TSF timestamps, which directly affect the mean errors in Table 4. Similar to the software PTP, one can use WNICs of the same model to avoid *TSF offset* (the host platforms may differ), otherwise, an

Network			PC (Client 1) → PC (AP 1) ↗ PC (Client 2) → PC (AP 2) ↘					
			CPU Load					
Rate	Size	PPS	0%		50%		100%	
6M	500B	0	0.24	0.72	0.17	0.82	-0.11	0.85
		100	-0.2	0.53	-0.2	0.69	0.32	0.75
48M	64B	1000	-0.03	0.72	-0.18	0.72	-0.16	0.67
			0.12	0.6	0.18	0.87	0.19	0.67
		5000	0.18	0.67	0.2	0.78	0.24	0.56
			A	B	C	D	E	F

Table 6: Cross-AP Hardware PTP Results (μ s). Notations are same as Table 2.

Factors	Acronyms	Meaning
Network & CPU Load	load	Network and CPU load are applied to (load) or not applied to (load) the client. When applying the load, we use the same configuration as cell 6-EE of Table 1, i.e., 48 Mbps, 64 Byte, 5000 PPS, 100% CPU.
Mobility	static* walk rotate	The mobility is introduced through holding the client to walk or rotating it, while keeping the AP static. The moving speed introduced by walking and rotating is around 1 m/s and 2 m/s.
Signal Strength	-30 dBm -55 dBm* -75 dBm	The received signal strength is adjusted by placing the client in different locations. The value is measured and reported by the WNIC hardware on the per packet basis.
Interference	none* 100 PPS 250 PPS	Two hosts are configured to transmit Wi-Fi packets (64 Byte @ 6 Mbps) with carrier sense disabled. Their transmissions not only occupy the channel but can also interfere with on-going PTP transmissions.
	default	Default configurations for comparison are denoted with *

Table 7: Measurement Acronyms.

offline calibration procedure is needed to measure and correct *TSF offset*. A typical solution (like the hardware offset in PTP Ethernet NICs) is to measure those model-dependent values with a reference clock, then the driver maintainer corrects them in the source code.

5.2.2 Wireless-Specific Impacting Factors

As a wireless synchronization approach, PTP over Wi-Fi may also be affected by wireless-specific factors. We study their impact by performing PTP under different mobility, signal strength, and interference levels. The error distributions are shown in Figure 6. In most cases, no obvious difference is observed from the default case, meaning that these factors do not affect PTP synchronization. This is because both hardware and software PTP depend on the accuracy of the timestamps of the PTP packets, but these wireless factors can not affect successfully-decoded packets³. Also, note that the neighboring interference significantly decreases the accuracy. This is because, due to the strong interference, PTP packets are jammed, postponed, and (mostly) dropped. As a result, the PTP algorithm does not have enough timestamps for clock adjustment, which significantly increases sync intervals and causes large jitters.

5.2.3 Cross-AP Synchronization

We conduct measurements of cross-AP synchronization (see Figure 3 (b)). Results are shown in Table 6. Compared with the Single-AP situation in Table 4, the mean error is similar. This is because when APs are synchronized boundary clocks,

³Depending on the implementation, TSF timestamps might be related to the PHY-layer frame synchronization, which is affected by channel conditions. However, the error should be within one cyclic prefix, i.e., 0.8 μ s in 802.11a.

Network		PC (Client 1) → PC (AP 1) ↗ Jetson (Client 2) → PC (AP 2) ↘						
		CPU Load						
Rate	Size	PPS	0%		50%		100%	
6M	500B	0	0.14	0.96	0.12	1.07	0.02	1.19
		100	0.43	1.04	0.52	0.82	0.8	0.74
48M	64B	1000	0.38	0.85	-0.53	1.05	0.14	1.08
			0.06	0.54	-0.06	0.62	0.04	0.59
		5000	-0.01	0.92	0.05	1.03	0.11	0.94
			AA	BB	CC	DD	EE	FF

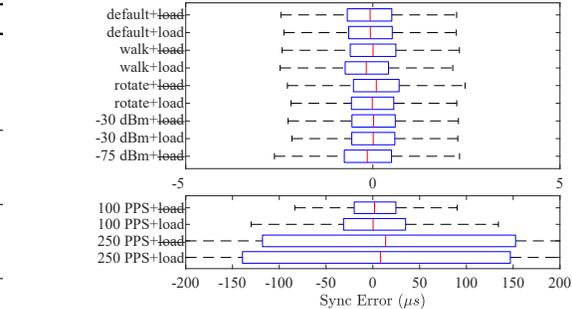


Figure 6: Sync Error Distribution v.s. Wireless Impacting Factors. The central mark indicates the median, and the left and right edges of the box indicate the 25th and 75th percentiles, respectively. Refer to Table 7 for the explanation of acronyms. There is no network load at -75 dBm case, as the SNR is not enough to decode 48 Mbps PHY rate packets.

clients synced to one AP are also synced to other APs and clients. The synchronization performance among clients is determined by the wired link and the Wi-Fi link. The former is guaranteed by Ethernet PTP (ns-level), while the latter has been studied in our previous measurements (μ s-level). However, jitters of the Cross-AP are more intense than that of the Single-AP. This is because Table 6 compares the clocks of client1 and client2 rather than clocks of client and AP. When two secondary clocks (hosted by client1 and client2) are synchronized to the same primary clock (hosted by AP1), their errors are independent random variables. The variance of their difference is the sum of their individual variances. This is a known limitation of the PTP boundary clock mode, whose error is accumulated along the sync chain [7].

5.3 Discussion - Hardware PTP

We discuss the potential improvement and power efficiency of hardware PTP, and then we compare it with software PTP.

Further Improvement. Hardware PTP syncs two hosts with high resolution system clocks (sub ns-level) via low-resolution (1 μ s) TSF clocks. In principle, the synchronization resolution is related to the stability, rather than the resolution of TSF clocks. A finer resolution might be achieved by sending a burst of PTP measurement packets with a fixed inter-packet interval. It might be possible to infer or interpolate sub- μ s timestamps by analyzing the rounding/stepping patterns of TSF timestamps.

Power Efficiency. Hardware PTP does not need to timely activate the host system for software timestamping, since

HOST	WNIC	Software PTP	Hardware PTP
same	same	✓	✓
diff.	same	int. calib.	✓
same	diff.	hw calib.	hw calib.
diff.	diff.	int.+hw calib.	hw calib.
timestamp packets		ISR	hw support
read TSF counter		no need	hw support
power efficiency		similar	

Table 8: Software and Hardware PTP Requirements

the timestamps have been taken by the WNIC hardware and stored in the meta info associated with the packets. However, PTP traffic has a negative impact on the IEEE 802.11 PS mode, where the WNIC of the client is aggressively turned down into a low power mode when there is no traffic. The AP buffers packets for the PS client and sends notifications through the beacons. The client WNIC wakes up at every beacon to check if there are packets for it, if there are, the WNIC will heuristically stay active for a certain period. As a result, the PTP traffic stimulates the WNIC periodically, making the PS mode deficient. We measure the impact with a power meter. When there is no traffic, the PS mode can save 0.37 W. When the PTP (4 PPS) is enabled, the PS can only save 0.15 W. Considering the overall standby power of the Jetson board is only 2 to 3 W, the trade-off between synchronization accuracy and power efficiency must be carefully considered when applying PTP to low-power applications.

On the other hand, when the PS is enabled, PTP packets will be delayed for 30 to 100 ms, but the delay does not affect the synchronization performance. This is because the timestamp of a buffered packet is taken when it is sent off to the air and when it is correctly received by the client.

Hardware v.s. Software PTP. From Table 2 and 4, both hardware and software PTP implementations can achieve μ s-level accuracy with COTS Wi-Fi devices, which we believe is sufficient for most time sensitive mobile applications. Besides, they inherit the benefit of PTP and can extend the synchronization range across multiple APs. To make the choice between them, we summarize their requirements in Table 8. When using different hosts for the primary and secondary clocks, software PTP needs to calibrate the interrupt responsiveness (int.calib. see §4.2.3). When using different WNICs, they both need offline calibration to correct hardware offsets (hw calib. see §4.2.3 and §5.2.1). We note that to support hardware PTP, the WNIC should be able to timestamp packets with its TSF counter and also provide the counter reading interface (see §5.1). While TSF is a mandatory feature of IEEE 802.11, these features are not, hence they might not be available on COTS WNICs. In short, if there are supported WNICs, hardware PTP is a convenient and accurate choice for Wi-Fi synchronization. Otherwise, software PTP is a more general and flexible choice with different levels of accuracy requiring different levels of calibration.

6 Other Wi-Fi Synchronization Protocols

This section compares PTP with TSF and FTM.

TSF synchronization is automatically finished by the WNIC hardware. The AP broadcasts the value of its TSF counter in beacon frames, according to which its client WNICs reset their TSF counters. Since TSF is originally designed for timing functions of Wi-Fi protocol, the TSF counter by default is not related to the host system clock. Therefore, we measure the difference between the TSF counters of the AP and the client to estimate the synchronization error. Their TSF counters are read and sandwiched by synchronized Ethernet PTP clocks like Figure 5. The time offset between the client and AP counter is calculated and shown in Table 9.

Similar to the hardware PTP implementation, errors of TSF synchronization are independent of the host system and the system load. There is a bias error of -4μ s and the jitter is larger. The bias error is attributed to different TSF offsets of the Tx and Rx chain, *i.e.*, $\delta_{Tx} \neq \delta_{Rx}$. Bias is a common error source of single-way synchronization methods. The jitter is caused by the TSF counter jumps caused by the frequency drift. Additionally, TSF cannot be directly used to sync clocks in the cross-AP situation. We note that while the above problems are not fundamental, for example, the bias error can be calibrated, a complete solution would probably just lead to our hardware PTP implementation, which makes use of the two-way PTP measurement to factor the offset off and leverages existing PTP interface to sync clocks in a scalable manner.

FTM/TM is the latest feature of the IEEE 802.11 standard. They use dedicated WNIC hardware for timestamping. FTM and TM differ in resolution. TM is at the level of 10 ns, while FTM is ps-level. FTM's high time resolution is designed for measuring the time-of-flight (ToF) of radio waves for ranging. Another goal of TM/FTM (IEEE 802.11as Clause 12 [2]) is to extend the LAN PTP synchronization to Wi-Fi devices, targeting the same problem as our PTP implementations.

Although some COTS WNICs support FTM for localization [27], they rarely expose the synchronization interface. After extensive tries, the only feasible way we found (by referring to the PTP daemon `gptp` [8] managed by Intel) to enable synchronization of FTM WNICs is: 1. using Intel FTM WNICs (Intel 8260 AC) with 2. proprietary Windows driver [11] 3. in the Wi-Fi direct mode (Ad-Hoc mode). We also note that even though it does work, the running protocol is TM rather than FTM (they differ in time resolution), which is observed through packet sniffing.

Given the above hardware and software, we are still not able to measure the accuracy of TM/FTM as Windows 10 does not natively support hardware PTP and Windows Server does not support the WNIC driver, thus the ground truth cannot be obtained. However, the std of the sync error can be measured and is shown in Table 10. The std is around 0.4μ s, which is close to our hardware PTP based on the TSF counter, but

Network			PC (Client) $\xrightarrow{\text{sync to}}$ PC (AP)					
			CPU Load					
Rate	Size	PPS	0%		50%		100%	
/	/	0	-3.78	1.09	-3.62	0.96	-3.67	1.13
6M	500B	100	-3.67	1.16	-3.53	1	-3.65	0.96
		1000	-3.91	0.99	-3.79	0.98	-3.71	0.99
48M	64B	1000	-3.76	1.14	-3.88	0.85	-3.67	1.01
		5000	-3.8	0.91	-3.72	0.98	-3.82	1.12
			A	B	C	D	E	F

Jetson (Client) $\xrightarrow{\text{sync to}}$ PC (AP)								
			CPU Load					
			0%		50%		100%	
1	4.08	1.06	-3.83	0.93	-3.67	1.32		
2	-3.86	1.09	-3.49	1.16	-3.71	0.94		
3	-3.74	1.05	-3.83	1.18	-3.82	1.15		
4	-3.99	1.15	-4.07	1.2	-3.78	0.99		
5	-3.92	0.92	-4	1.09	-3.85	1.04		
6	-3.87	1.07	-3.79	0.96	-3.71	1.12		
			AA	BB	CC	DD	EE	FF

Table 9: TSF Synchronization Results. Values are in unit of μs . Notations are same as Table 2.

Network	CPU Load			std
	0%	50%	100%	
0%	0.39	0.44	0.41	
50%	0.36	0.36	0.34	
100%	0.37	0.38	0.39	

Table 10: TM Results. Values are std of estimated synchronization error in unit of μs . CPU load and network load (represented by channel occupancy) are generated by CPUSTRES and PSPING from Sysinternals [19]

this value is actually very large compared with its 10 ns-level timestamp resolution. The reason is unknown but it is probably caused by the incomplete implementation or being intentionally disabled/hidden by the Intel firmware.

We believe FTM and TM provide an ideal solution for Wi-Fi synchronization, but it seems its primary goal is positioning and the synchronization feature is publicly unavailable right now. This is probably because FTM is only an optional feature of IEEE 802.11, or simply because of market reasons.

7 Related Work

Due to the importance of synchronized time in distributed systems [30], network synchronization protocols have been extensively discussed [42]. In a small network scale like LANs, the network delay between nodes is stable under prioritized switching [47] and the packets can be accurately timestamped by NIC hardware. These facts guarantee the performance of LAN PTP, and in practice it achieves nanosecond accuracy with commercial Ethernet NICs [25]. It is natural to extend PTP to wireless LANs due to the synchronization demand of mobile systems [44]. Corresponding practices started a decade ago and a comprehensive survey is given by [34]. We next review software and hardware PTP separately.

For hardware PTP, existing work focuses on improving timestamp accuracy and precision. The challenge is that wireless channels bring uncertainties to the received RF signals, making it hard to determine the accurate start of the packet. Modern OFDM-based and wideband PHYs allow the accuracy to reach several ps. However, most research work is based on laboratory prototypes [22, 24, 28] and commercial FTM WNICs are mainly for ranging and positioning purposes [27]. As a result, the performance of hardware PTP with COTS Wi-Fi devices remain unknown. Our work makes hardware PTP available on COTS devices and then characterizes its performance.

Our hardware PTP implementation is based on the TSF counter, so it is related to TSF-based synchronization ap-

proaches. RF-WiFi [45] makes use of TSF interrupts to notify the host systems so that they can take synced actions. Mahmood *et al.* study the IEEE 802.11 Timing Advertisement feature [36], which uses beacon frames to convey AP’s system clock to clients. However similar to TSF, it is single-way and thus has a bias. We have not encountered any works which make use a TSF counter as a hardware PTP clock.

For software PTP, the majority of existing work was conducted ten years ago with IEEE 802.11b Atheros WNICs. As a result, most insights no longer sufficient to light the current situation due to the rapid evolution of mobile systems. For example, the PCI DMA latency [33] becomes a minor issue since newer WNICs use PCIe [40]. The TSC clock is stable and independent of the core frequency [35] due to the adoption of invariant TSC in modern processors [21]. The impact of the CPU and network load is studied by existing work [26, 33], but our measurement reveals that they are not major factors.

Existing discussions on CPU idle power management focus on server platforms [43]. Our work reveals its impact on mobile platforms and on synchronization performance. Further, we have not noticed any discussion on interrupt mitigation under the context of Wi-Fi networks. As far we know, Atheros 802.11b WNICs does not have this feature, and hence past work based on old WNICs is able to achieve stable software PTP performance even without realizing this factor.

8 Conclusion

Accurate time synchronization is a key enabler for many distributed control and sensing applications. In this work, we study the performance of PTP protocol in Wi-Fi networks through systemic implementation and rigorous evaluation. The in-depth discussion and open-source implementation render a useful reference for designing and adopting PTP in Wi-Fi networks.

Acknowledgements

We thank anonymous reviewers for their valuable comments. We sincerely thank our shepherd Lin Zhong. We thank Shauna Dalton for the proofreading. This work is supported (in part) by the ShanghaiTech Startup Fund, the Shanghai Sailing Program 18YF1416700, the “Chen Guang” Program 17CG66 supported by Shanghai Education Development Foundation and Shanghai Municipal Education Commission, and NSFC 62002224.

References

- [1] Ieee standard for a precision clock synchronization protocol for networked measurement and control systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pages 1–300, July 2008.
- [2] Ieee standard for local and metropolitan area networks - timing and synchronization for time-sensitive applications in bridged local area networks. *IEEE Std 802.1AS-2011*, pages 1–292, March 2011.
- [3] Ieee standard for information technology—telecommunications and information exchange between systems local and metropolitan area networks—specific requirements - part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)*, pages 1–3534, Dec 2016.
- [4] Adt-link pci express 3.0 x4 to m.2 nvme extension cable. <http://www.adt.link/product/R42U.html>, 2021.
- [5] Advanced configuration and power interface (acpi) specification, revision 6.3. https://uefi.org/sites/default/files/resources/ACPI_6_3_final_Jan30.pdf, May 2021.
- [6] Ccnt. <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0211h/Bihcgfcf.html>, 2021.
- [7] Configuring precision time protocol (ptp). https://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst9300/software/release/16-10/configuration_guide/lyr2/b_1610_lyr2_9300_cg/configuring_precision_time_protocol__ptp_.pdf, 2021.
- [8] gptp. <https://github.com/AVnu/gptp>, 2021.
- [9] Hikey970 evaluation board. <https://www.96boards.org/product/hikey970/>, 2021.
- [10] Intel 64 and ia-32 architectures software developer manuals. <http://www.intel.com/products/processor/manuals/>, 2021.
- [11] Intel proset/wireless software and drivers. <https://downloadcenter.intel.com/download/30280>, 2021.
- [12] Intel true view. <https://www.intel.com/content/www/us/en/sports/technology/true-view.html>, 2021.
- [13] Interrupt moderation of intel nics. <https://www.intel.com/content/www/us/en/support/articles/000005811/network-and-io/ethernet-products.html>, 2021.
- [14] Jetson xavier nx module and developer kit. <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/>, 2021.
- [15] The linux ptp project. <http://linuxptp.sourceforge.net>, 2021.
- [16] Odyssey x86 board. <https://www.seeedstudio.com/ODYSSEY-X86J4105800-p-4445.html>, 2021.
- [17] Ptp hardware clock infrastructure for linux. <https://www.kernel.org/doc/html/latest/driver-api/ptp.html>, 2021.
- [18] Wi-ptp. <https://github.com/Wireless-Lab/Wi-PTP>, 2021.
- [19] Windows sysinternals. <https://docs.microsoft.com/en-us/sysinternals/>, 2021.
- [20] S. Ansari, N. Wadhwa, R. Garg, and J. Chen. Wireless software synchronization of multiple distributed cameras. In *2019 IEEE International Conference on Computational Photography (ICCP)*, pages 1–9, 2019.
- [21] J. Coleman, S. Almalih, A. Slota, and Y.-H. Lee. Emerging cots architecture support for real-time tsn ethernet. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 258–265. ACM, 2019.
- [22] T. Cooklev, J. C. Eidson, and A. Pakdaman. An implementation of ieee 1588 over ieee 802.11 b for synchronization of wireless local area network nodes. *IEEE Transactions on Instrumentation and Measurement*, 56(5):1632–1639, 2007.
- [23] S. D’souza, H. Koehler, A. Joshi, S. Vaghani, and R. R. Rajkumar. Quartz: *time-as-a-service* for coordination in geo-distributed systems. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, SEC ’19*, page 264–279, New York, NY, USA, 2019. Association for Computing Machinery.
- [24] R. Exel. Clock synchronization in ieee 802.11 wireless lans using physical layer timestamps. In *2012 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication Proceedings*, pages 1–6, 2012.
- [25] B. Ferencz and T. Kovács házy. Hardware assisted cots ieee 1588 solution for x86 linux and its performance evaluation. In *2013 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication (ISPCS) Proceedings*, pages 47–52, 2013.

- [26] P. Ferrari, A. Flammini, S. Rinaldi, A. Bondavalli, and F. Brancati. Experimental characterization of uncertainty sources in a software-only synchronization system. *IEEE Transactions on Instrumentation and Measurement*, 61(5):1512–1521, 2012.
- [27] M. Ibrahim, H. Liu, M. Jawahar, V. Nguyen, M. Gruteser, R. Howard, B. Yu, and F. Bai. Verification: Accuracy evaluation of wifi fine time measurements on an open platform. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 417–427. ACM, 2018.
- [28] J. Kannisto, T. Vanhatupa, M. Hannikainen, and T. Hamalainen. Software and hardware prototypes of the ieee 1588 precision time protocol on wireless lan. In *2005 14th IEEE Workshop on Local and Metropolitan Area Networks*, pages 1–6, 2005.
- [29] J. Kannisto, T. Vanhatupa, M. Hännikäinen, and T. D. Hämäläinen. Precision time protocol prototype on wireless lan. In *International Conference on Telecommunications*, pages 1236–1245. Springer, 2004.
- [30] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
- [31] Z. Li, W. Chen, C. Li, M. Li, X.-Y. Li, and Y. Liu. Flight: Clock calibration using fluorescent lighting. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 329–340. ACM, 2012.
- [32] R. Lisovỳ, M. Sojka, and Z. Hanzálek. Pci express as a killer of software-based real-time ethernet. In *The 12th International Workshop on Real-Time Networks*, 2013.
- [33] A. Mahmood, R. Exel, and T. Sauter. Delay and jitter characterization for software-based clock synchronization over wlan using ptp. *IEEE Transactions on Industrial Informatics*, 10(2):1198–1206, 2014.
- [34] A. Mahmood, R. Exel, H. Trsek, and T. Sauter. Clock synchronization over ieee 802.11—a survey of methodologies and protocols. *IEEE Transactions on Industrial Informatics*, 13(2):907–922, 2017.
- [35] A. Mahmood, G. Gaderer, H. Trsek, S. Schwalowsky, and N. Kero. Towards high accuracy in ieee 802.11 based clock synchronization using ptp. In *2011 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, pages 13–18, 2011.
- [36] A. Mahmood and T. Sauter. Limitations in implementing wireless clock synchronization using the timing advertisement approach from ieee 802.11. In *2016 IEEE International Instrumentation and Measurement Technology Conference Proceedings*, pages 1–6. IEEE, 2016.
- [37] S. K. Mani, R. Durairajan, P. Barford, and J. Sommers. Mntp: enhancing time synchronization for mobile devices. In *Proceedings of the 2016 Internet Measurement Conference*, pages 335–348. ACM, 2016.
- [38] P. Membrey, D. Veitch, and R. K. Chang. Time to measure the pi. In *Proceedings of the 2016 Internet Measurement Conference*, pages 327–334. ACM, 2016.
- [39] D. L. Mills. Improved algorithms for synchronizing computer network clocks. *IEEE/ACM transactions on Networking*, 3(3):245–254, 1995.
- [40] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 327–341, 2018.
- [41] H. Rahul, H. Hassanieh, and D. Katabi. Sourcesync: a distributed wireless architecture for exploiting sender diversity. *ACM SIGCOMM Computer Communication Review*, 41(4):171–182, 2011.
- [42] J. Ridoux, D. Veitch, and T. Broomhead. The case for feed-forward clock synchronization. *IEEE/ACM Transactions on Networking (TON)*, 20(1):231–242, 2012.
- [43] R. Schöne, D. Molka, and M. Werner. Wake-up latencies for processor idle states on current x86 processors. *Computer Science-Research and Development*, 30(2):219–227, 2015.
- [44] K. B. Stanton. Distributing deterministic, accurate time for tightly coordinated network and software applications: ieee 802.1as, the tsn profile of ptp. *IEEE Communications Standards Magazine*, 2(2):34–40, 2018.
- [45] Y.-H. Wei, Q. Leng, S. Han, A. K. Mok, W. Zhang, and M. Tomizuka. Rt-wifi: Real-time high-speed communication protocol for wireless cyber-physical control applications. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 140–149. IEEE, 2013.
- [46] M. Weiss. Telecom requirements for time and frequency synchronization. In *National Institute of Standards and Technology (NIST), USA*. [Online]: www.gps.gov/cgsic/meetings/2012/weiss1.pdf, 2012.
- [47] Z. Yang, J. Zhang, K. Tan, Q. Zhang, and Y. Zhang. Enabling tdma for today’s wireless lans. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1436–1444. IEEE, 2015.

AUTO: Adaptive Congestion Control Based on Multi-Objective Reinforcement Learning for the Satellite-Ground Integrated Network

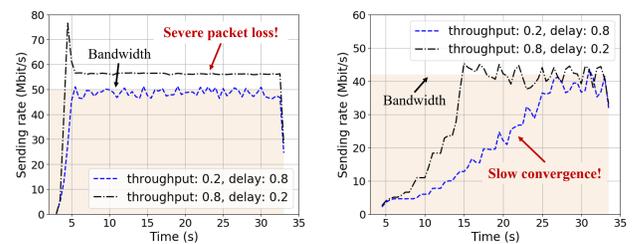
Xu Li[†], Feilong Tang^{†*}, Jiacheng Liu[†], Laurence T. Yang[‡], Luoyi Fu[†], Long Chen[†]
[†] Department of Computer Science, Shanghai Jiao Tong University
[‡] Department of Computer Science, St. Francis Xavier University

Abstract

The satellite-ground integrated network is highly heterogeneous with diversified applications. It requires congestion control (CC) to achieve consistent high performances in both long-latency satellite networks and large-bandwidth terrestrial networks and cope with different application requirements. However, existing schemes can hardly achieve these goals, for they cannot balance the objectives of CC (i.e., throughput, delay) adaptively and are not objective-configurable. To address these limitations, we propose and implement a novel adaptive CC scheme named AUTO, based on Multi-Objective Reinforcement Learning (MORL). It is *environment-adaptive* by training a MORL agent and a preference adaptation model. The first can generate optimal policies for all possible preferences (i.e., the relative importance of objectives). The latter automatically selects an appropriate preference for each environment, by taking a state sequence as input to recognize the environment. Meanwhile, AUTO can satisfy diversified application requirements by letting applications determine the input preference at will. Evaluations on emulated networks and the real Internet show that AUTO consistently outperforms the state-of-the-art in representative network environments and is more robust to stochastic packet loss and rapid network changes. Moreover, AUTO can achieve fairness against different CC schemes.

1 Introduction

To achieve global coverage and meet the excessive custom demand for data access, the satellite-ground integrated network has attracted intensive research interest [1]. It is highly heterogeneous where a node can send data to different peers through both satellite networks located at different orbits, and ground networks including the Internet and cellular networks [2]. To achieve consistent high user experiences in all these environments with widely varying packet loss rates, round-trip time (RTT) and available bandwidth, *adaptive Congestion Control (CC)* is one of the key technologies [3]. Basically,



(a) Inter-datacenter network

(b) Satellite network

Figure 1: Sending rates of two CC schemes that adopts two different preferences. A fixed preference can only achieve high performance in specific types of environments.

it determines sending rates, convergence speeds, and the reactivity to network fluctuations of underlining flows. It has two competing objectives, i.e., optimizing link utilization (high throughput) while avoiding congestion (low queuing delay) [4].

The two objectives have to be balanced adaptively in different environments [5], since a fixed preference (i.e., the relative importance of the objectives) can only achieve high performance in specific types of environments. To illustrate this phenomenon, we conduct an experiment by first training two Reinforcement Learning (RL) based CC schemes using the same training ground with two different preferences. We evaluated their performances in the emulated inter-datacenter network and the satellite network WINDs [6] based on Pantheon [7]. The former is high-speed and has a short buffer while the latter has a long latency. As shown in Figure 1, putting higher weight on “throughput” achieves good performance in the satellite network but meanwhile suffers from severe packet loss in the ground network. The main reason is that a saturated buffer only has little punishment on the final reward and the trained model cannot detect the congestion. In contrast, putting higher weight on “delay” suffers from slow convergence in the satellite network because it is too sensitive to delay variation.

Meanwhile, a CC schemes for the integrated network

*Feilong Tang is the corresponding author of this paper.

will serve diversified applications at the same time. Delay-sensitive applications such as online meetings require higher weight to be put on “delay”, while throughput-sensitive applications such as file synchronization require higher weight to be put on “throughput”. To cope with all these requirements, the CC scheme should be able to let applications determine the preference at will.

Lots of CC schemes have been proposed in recent years, which can be classified into *heuristic-based* or *learning-based* schemes. Unfortunately, they cannot achieve the consistent high performance in all environments nor cope with diversified application requirements [7]. We analyze the reasons in what follows.

Heuristic-based methods hard-wire actions with predefined events or signals (e.g., packet loss, delay variation) based on the analysis of network characteristics. However, the analysis is not suitable for all kinds of network environments [8]. For example, *loss-based* methods such as TCP Cubic [9] take packet loss as the sign of congestion and is not suitable for unreliable lossy scenarios, while the *delay-based methods* such as Copa [4] don’t work well in the short buffer scenarios for the delay fluctuates in a very small range [10]. Meanwhile, the hard-wire mapping fundamentally limits the objective configurability.

Existing learning-based methods can only achieve high performance in specific network environments [5, 8]. One of the main reasons is that they convert the objectives into a single reward function (RL) [5, 11–15] or utility function (online learning) [16–18] by introducing fixed empirical preferences and determine actions accordingly. Although training a series of models with different preferences to cope with different environments may alleviate this problem, the lack of an environment recognition method makes them unable to switch models adaptively.

Based on above analyses, we in this paper propose an *Adaptive congestion control method based on mULTI-Objective reinforcement learning*, abbreviated as AUTO. It first divides time into consecutive monitoring intervals and formulates the CC as an extended version of the multi-objective Markov decision process. It then trains a *MORL agent* and a *preference adaptation model*, based on which it adjusts the sending rate at the end of each interval. Compared with existing approaches, it has following characteristics.

Firstly, AUTO is *environment-adaptive*. It automatically balances the two objectives in different environments where the preference adaptation model can adaptively select an environment-adaptive preference for users, by taking a state sequence as input to recognize the network environment. Meanwhile, the policy agent can generate optimal policies for all possible network states and preferences. Combining the two components, AUTO achieves the consistent high performance in different environments.

Secondly, AUTO is *objective-configurable*. Since the MORL agent only needs to be trained once and can deal

with any input preferences, AUTO can cope with diversified application requirements.

Moreover, AUTO is *competitive-configurable*. The input preference determines the AUTO’s delay-sensitivity and further influence its competitiveness against other flows. Therefore, AUTO can achieve fairness against different competing CC schemes and allows users to explicit adjust the priority of different flows, by adjusting the preference.

Main contributions are summarized as follows.

1. We propose an efficient and practical *Multi-Objective Reinforcement Learning (MORL) framework*. It trains a single MORL agent that can recover optimal policies for all possible preferences.
2. We propose a novel *environment-adaptive preference selection method*. We first propose a policy similarity model, based on which we find the best preference for each training environment by comparing with the expert policy. Then we train a preference adaptation model to automatically select preferences.
3. We propose and implement AUTO based on the above framework and method. It is easy to deploy for it requires sender-only modification. It consistently outperforms than the state-of-the-art in representative scenarios and is more robust to network changes and packet loss. Moreover, it can achieve fairness against other CC methods.
4. We develop an emulation-based training suite Pantheon-Gym for MORL-based CC. It eases the training by providing standard OpenAI Gym interfaces for researchers.

2 Related Work

We classify related work on learning-based CC methods into the following three categories.

RL-based methods train a policy model with the goal of maximizing a reward function. QTCP [13] adopted the Q-Learning framework and formulate the reward function as the proportional fairness [19] of throughput and delay. RL-TCP [12] adopted a SARSA [20] based RL framework and added the packet loss rate in the reward function. Their actions specify how to change the congestion window in response to variations in the network environments and they are trained on simulators NS2 [21] and NS3 [22] respectively. Aurora [5] formulate the reward function as the weight sum of throughput and delay. Meanwhile, it implements a simulation-based gym environment. To train the agent on real network environments, authors in [14] adopt the same reward function and designed Park, which is an open platform for learning-augmented computer systems. Considering the delayed action phenomenon, an asynchronous RL framework called MVFST-RL was proposed in [11]. To combine reinforcement learning and traditional CC schemes, TCP-RL [15] took different traditional CC schemes as the action space and trained the model through emulations built by the Linux tool. To solve the CC problem

for multi-path TCP, authors in [23] proposed SmartCC, which calculates the reward according to throughput, latency, and delay jitter. Authors in [24] proposed DRL-CC, which adopts the goodput as rewards. To solve the centralized congestion control problem, Iroko [25] formulate the reward function as a function of the bandwidth utilization and the latency of all links. Its action is to regulate the sending rates of all nodes. However, the trained policy model can only be applied to the fixed network topology and is intractable for large networks.

Online learning-based methods try to fine-tune the sending rate and decide the adjustment direction according to a utility function and they also split the time into consecutive monitoring intervals. PCC-Allegro [16] formulates the utility function as a function of the sending rate and the loss rate, while Vivace [17] puts delay into consideration on this basis. PCC Proteus [18] defines two kinds of utility functions. It can behave as a primary protocol for primary flows or an effective “scavenger” that only utilizes the residue bandwidth of primary flows. Compared with the RL-based methods, online learning-based methods cannot learn the prevailing network regularities and have been shown to have poorer performance than RL-based methods [5].

Inverse RL-based methods seek to avoid the empirical preference setting by finding a reward function from the expert demonstrations. Indigo [7] trained a model to adjust the sending rate every 10 ms by directly imitating expert behaviors. This kind of method is not objective-configurable and cannot cope with different application requirements. Meanwhile, both the RL-based and inverse RL-based methods cannot achieve fairness against other flows [5], since their competitiveness is not configurable and strongly related to the training environments.

To cope with the heterogeneity of the integrated network, we in this paper propose a new kind of learning-based method, i.e., *MORL-based method*. It addresses the limitations of existing methods and is environment-adaptive, objective-configurable, and competitive-configurable. Thus, it can achieve consistent high user experience for flows passing through different environments and with different performance requirements.

3 Multi-objective Reinforcement Learning based Congestion Control

In this section, we propose practical methods for training the MORL agent that can generate optimal policies for all possible preferences.

3.1 Congestion Control Formulation

In order to apply the MORL framework, we first formulate the congestion control as a *Multi-Objective Markov Decision Process (MOMDP) with delayed actions*. Formally, it can be described by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \Omega, \gamma, f_\Omega)$, where

- $\mathcal{S} \subseteq \mathbb{R}^n$ is the continuous state space;
- \mathcal{A} is the discrete action set;
- \mathcal{P} is the Markovian transition model;
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}^2$ is the set of two reward functions and corresponds to the two objectives for CC;
- $\Omega \subseteq \mathbb{R}^2$ is the preference space;
- $\gamma \in [0, 1)$ is the discount factor;
- f_Ω is the set of preference functions that convert the reward vector into one scalar according to the chosen preference $\omega \in \Omega$.

In this work, preference $\omega \in \Omega$ is a vector of two values, i.e., $\omega = [\omega_t, \omega_d]$ is the weight on throughput and delay respectively. Meanwhile, we consider functions in f_Ω are linear functions, i.e.,

$$f_\omega(\mathcal{R}(s, a)) = \omega^T \mathcal{R}(s, a), \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \quad (1)$$

In the MOMDP, a policy π is associated to an expected return for a given preference $\omega \in \Omega$, which is denoted as

$$J_\omega^\pi = \mathbb{E}_{s_t \sim \mathcal{P}, a_t \sim \pi} \left[\sum_{t=0}^T \gamma \cdot f_\omega(\mathcal{R}(s_t, a_t)) \right] \quad (2)$$

where $s_t \sim \mathcal{P}$ means state s_t is sample from \mathcal{P} and $a_t \sim \pi$ means action a_t is selected according to policy π .

The goal for solving MOMDP is to find the set of Pareto-optimal policies for all possible preferences, which is depicted as

$$\Pi^* = \{\pi | \exists \omega \in \Omega, \nexists \pi' : J_\omega^{\pi'} > J_\omega^\pi\}. \quad (3)$$

MOMDP with delayed actions is an extension of the MOMDP, which means that actions won't take effect immediately and therefore the state s_{t+1} is not only influenced by action a_t and state s_t , but also influenced by the previous actions and states. It is caused by two reasons. Firstly, the apply of the action is delayed due to the policy lookup time [11]. Since the environment is running asynchronously, it would keep transmitting data based on the old action a_{t-1} during the policy looking up. Secondly, the state observation is delayed since the states are updated based on packet ACKs. After action a_t is applied, it takes about $0.5 \cdot \text{rtt}_t$ for receiver to receive the packets sent based on a_t . It takes another $0.5 \cdot \text{rtt}_t$ to response the ACKs. Thus, the observed network state is influenced by the action made rtt_t before.

In what follows, we describe the formulation of state space, reward functions, the action set and the Monitoring Interval (MI) in detail.

State space considering delayed actions (\mathcal{S}). To accurately identify the network congestion without explicit network information, we select the following features to model network states. To further improve the robustness and generalization of the trained model, we try to use relative values instead of absolute values such as latency and sending rate.

- **Latency Ratio** r_{lat}^t [19]. Latency ratio is the ratio of the current MI's mean latency to minimum observed mean latency, which is one of the most important features to detect congestion.
- **Sending Ratio** r_{send}^t [5]. Sending ratio is the ratio of the number of sent packets to the number of acknowledged packets. It helps agents adjust the sending rate.
- **Packet Loss Rate** r_{loss}^t . The r_{send}^t is influenced by the inflight ACKs. To prevent the agent from overestimating the congestion level, we add r_{loss}^t in the state.
- **Latency Gradient** g_{lat}^t [17]. Latency gradient is the derivative of latency with respect to time. The agent can infer that the sending rate is greater than the capacity if $g_{lat}^t > 0$.
- **Action** a_t . Considering that the state is influenced by the previous actions (the delayed action phenomenon), the action history should be put in the state to fasten convergence [11].

The delayed action phenomenon makes the reinforcement learning more challenging, since it is hard for agents to catch the real transition for the MOMDP. Therefore, we let each state be a history of the network statistics so that the agent can capture the influence of previous action on future states. We use h to represent the history length and the state $s_t \in \mathcal{S}$ can be formulated as

$$s_t = \{r_{lat}^{t-i}, r_{send}^{t-i}, r_{loss}^{t-i}, g_{lat}^{t-i}, a_{t-i} \mid i \in [0, h]\}. \quad (4)$$

Multi-Objective Reward Functions (\mathcal{R}). The goal of the CC is to achieve both high throughput and low queuing delay. Correspondingly, we adopt the average throughput, denoted as throughput_t , and the negative mean RTT as reward functions, i.e.,

$$\mathcal{R}(s_t, a_t) = [\text{throughput}_t, -\text{rtt}_t]. \quad (5)$$

RTT-adaptive monitoring interval. In AUTO, the state and the rewards are calculated at the end of each MI. A short MI is not enough for capturing the real transition in long-RTT scenarios, while setting the MI too long reducing the real-time responsiveness to the network dynamics in short-RTT scenarios. To cope with the heterogeneous network environment and considering that each action takes about $0.5 \cdot \text{rtt}_{t-1}$ to take effects, we adopt the RTT-adaptive MI and set the length of each MI to

$$|\text{MI}_t| = \min\{0.5 \cdot \text{rtt}_{t-1}, 1.5 \cdot \text{rtt}_r^{\min}\} \quad (6)$$

where rtt_r^{\min} is the observed minimum RTT. It guarantees that the MI is longer enough and meanwhile let CC be able to quick react to congestion.

Action set (\mathcal{A}). In our proposed CC scheme, each action corresponds to an adjustment on the sending rate. To adapt to heterogeneous environments with variant bandwidth, we seek

to adjust the sending rate proportionally and adopt an action set containing seven discrete values:

$$\mathcal{A} = \{\div 2, \div 1.3, \div 1.1, \times 1, \times 1.1, \times 1.3, \times 2\} \quad (7)$$

where “ $\div x$ ” means “dividing the sending rate by x ”, and “ $\times x$ ” means “multiplying the sending rate by x ”.

Since the reward function is the combination of throughput and delay, the agent will observe the equivalently bad rewards when the sending rate raises to 50x or 100x bandwidth and the buffer queue is saturated. In the training stage, bonding the sending rate to reasonable limits will produce the same transitions and meanwhile decrease CPU overheads caused by packet generation and sending. Thus, we set the sending rate in MI_t , denoted as sr_{t+1} , to

$$sr_{t+1} = \min(\text{apply}(sr_t, a_t), 500\text{Mbps}) \quad (8)$$

where $\text{apply}(sr_t, a_t)$ is a function that applies action a_t on sending rate sr_t .

3.2 Parallel MORL Training Framework

The goal of the MORL agent is to achieve consistent high performance in diverse environments under different preferences. Since the optimal policy in these scenarios varies, the MORL agent has to be trained in a series of environments with different network characteristics and input preferences. It results in the *catastrophic forgetting problem* [26] and *extended training time*, where catastrophic forgetting means that the knowledge for previously learned environment is abruptly destroyed by the training in the new environment.

To cope with these issues, we utilize an efficient parallel MORL training framework as shown in Figure 2. Basically, it is an extended version of the Asynchronous Advantage Actor-Critic framework [27] and has three key points. Firstly, it is *asynchronous*, which means multiple agents are used to train actors in parallel and updates the global parameters periodically. To deal with the catastrophic forgetting problem, we train the agent in different environments simultaneously, by letting each agent interacts with a specific type of environment (e.g., long latency, low bandwidth, high throughput weight). Secondly, it is based on the actor-critic paradigm [28]. In the training phase, the actor updates the policy parameters in the direction suggested by the critic while the critic updates the value function parameters. Thirdly, it uses the *advantage* function instead of the raw value of an action [29], which helps us better compare the actions for a given state and makes the training process more stable.

The adopted framework greatly improves learning efficiency. On the one hand, it fastens the interaction through parallel training. On the other hand, it improves the sample efficiency by adopting the off-policy RL approach, which reuses any past episodes through the experience replay mechanism. Furthermore, it improves the exploration efficiency since the

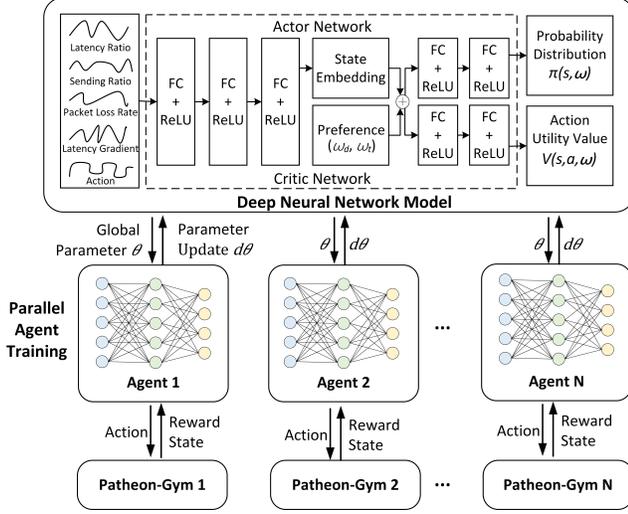


Figure 2: MORL Training Framework.

adopted off-policy brings better explorations for sample collection, by following a behavior policy that is different from the target policy.

Deep Neural Network Model. As shown in Figure 2, we use two deep neural networks with state s and preference ω as input and $2|\mathcal{A}|$ Q-values as output. The first is the actor network, which outputs the probability distribution over the action space. The second is the critic network, which outputs the utility value of the action. The parameters in two networks are denoted as θ_π and θ_v , respectively.

The two networks share three fully connected layers, each of which uses a rectified linear activation unit (ReLU) for feature extraction from raw inputs. The extracted features are then concatenated with the preference value and fed into different fully connected layers for output.

3.3 Agent Training Strategies

To train the MORL agent with multiple reward functions, we update the neural network model according to a generalized version of bellman equation [30]

$$\mathbf{Q}^\pi(s, a, \omega) = \mathbb{E}_{s' \sim P} \left[\mathcal{R}(s, a) + \gamma \mathbb{E}_{a' \sim \pi} [\mathbf{Q}^\pi(s', a', \omega)] \right] \quad (9)$$

where \mathbf{Q} is the Multi-Objective Q-value (MOQ-value) function. Its key idea is to use the vectorized value function to perform envelope updates. It allows our method to quickly align one preference with optimal rewards and trajectories that may have been explored under other preference. Thus, it can learn a single parametric representation for optimal policies over the whole preference space.

Based on the above equation, we can calculate the target for a given transition (s, a, s', \mathbf{r}) at step k by

$$\mathbf{y} = \mathbb{E}[\mathbf{r} + \gamma \arg_{\omega'} \max[\omega'^T \mathbf{Q}(s', a, \omega'; \theta_k)]] \quad (10)$$

where s and s' are the previous state and the current state, a is the adopted action, \mathbf{r} is the reward vector, ω' is sampled according to a fixed distribution, θ_k is the network parameters at step k and $\mathbf{Q}(\cdot; \theta)$ is the MOQ-value function parameterized by θ . Then, we can use the mean square error (MSE) between the prediction returned by the neural networks and the target,

$$L^A(\theta) = \mathbb{E}_{s, a, \omega} \left[\|\mathbf{y} - \mathbf{Q}(s, a, \omega; \theta)\|_2^2 \right] \quad (11)$$

3.3.1 Homotopy optimization

Unfortunately, directly optimizing L^A is challenging in practice since it is non-smooth considering that the optimal frontier contains a large number of discrete solutions. To solve this problem, we adopt homotopy optimization [31] similar to [30]. To be specific, we construct an auxiliary loss function L^B that directly optimize the scalarized Q value as,

$$L^B(\theta) = \mathbb{E}_{s, a, \omega} [\|\omega^T \mathbf{y} - \omega^T \mathbf{Q}(s, a, \omega; \theta)\|]. \quad (12)$$

Thus, the overall loss function is formulated as

$$L(\theta) = (1 - \lambda) \cdot L^A(\theta) + \lambda \cdot L^B(\theta) \quad (13)$$

where λ slowly increases from 0 to 1 in the training process and shifts the loss function from L^A to L^B .

Finally, we can calculate the stochastic gradient of network parameters by

$$d\theta_\pi = \frac{1}{N_\omega N_\tau} \sum_{i, j} T_{ij} \quad (14)$$

$$d\theta_v = (1 - \lambda) \cdot \nabla_{\theta_v} L^A(\theta_v) + \lambda \cdot \nabla_{\theta_v} L^B(\theta_v) \quad (15)$$

where N_ω and N_τ are the minibatch sizes for sampling transitions and preferences respectively and

$$T_{ij} = [\omega_i^T (\mathbf{V}_{ij} - \mathbf{V}(s_j, \omega_i; \theta_v))] \nabla_{\theta_\pi} \log \pi(a_j | s_j, \omega_i; \theta_\pi).$$

Here, $\mathbf{V}(s_j, \omega_i; \theta_v)$ is the baseline for calculating the advantage [29] and \mathbf{V}_{ij} is the estimated optimal value, which can be calculated by

$$\mathbf{V}_{ij} = \begin{cases} \mathbf{r}_j & \text{if done} \\ \mathbf{r}_j + \gamma \arg_{\omega' \in W} \max \omega_i^T \mathbf{V}(s_{j+1}, \omega'; \theta) & \text{o.w.} \end{cases} \quad (16)$$

3.3.2 Early termination trick

For training efficiency, the *needle-in-the-haystack problem* [14] has to be solved, which is described as follows. In the exploration stage of the training, the agent performs random walks and may be trapped in congestion states, i.e., when the sending rate is above the available network bandwidth and the buffer is saturated, the agent can only observe equivalently bad rewards (which is the combination of throughput and latency). It provides meaningless gradients and results in extremely inefficient training.

To solve this problem and stabilize the training process, we propose a simple but useful method called *early termination trick*. The basic idea is to increase the training step length according to the episode index. In this work, we adopt a linear function to calculate the termination index of each training step, which is depicted as

$$f_{et}(i) = \delta_a + (i \bmod \delta_b) \quad (17)$$

where δ_a is the initiate step length and δ_b is the step growth speed.

Algorithm 1: Agent Training Algorithm

Input: Minibatch sizes N_τ and N_ω ; network parameters θ_π and θ_v ; λ for homotopy optimization; δ_a and δ_b in the early termination trick.

Output: Trained parameters θ_π and θ_v .

- 1 Initialize replay buffer $\mathcal{D}_\tau = \{\}$.
 - 2 Set $\mathcal{D}_\omega = U\{\omega_r + \omega_d = 1 | \omega_r \in (0, 1), \omega_d \in (0, 1)\}$.
 - 3 **for** $episode = 1, \dots, M$ **do**
 - 4 Synchronize parameters $\theta'_v = \theta_v$ and $\theta'_\pi = \theta_\pi$.
 - 5 Sample a preference $\omega \sim \mathcal{D}_\omega$.
 - 6 **for** $t = 0, \dots, N - 1$ **do**
 - 7 **if** $t \geq f_{et}(episode)$ **then**
 - 8 **break**
 - 9 Calculate state s_t according to Eq. (4).
 - 10 Sample an action $a_t \sim \pi(a_t | s_t, \omega; \theta'_\pi)$.
 - 11 Get $(\mathbf{r}_t, s_{t+1}, done)$ from the environment.
 - 12 Set $\mathcal{D}_\tau = \mathcal{D}_\tau \cup (s_t, a_t, \mathbf{r}_t, s_{t+1})$.
 - 13 **if** $|\mathcal{D}_\tau| \geq N_\tau$ **then**
 - 14 Sample $(s_j, a_j, \mathbf{r}_j, s_{j+1}) \sim \mathcal{D}_\tau$.
 - 15 Sample $W = \{\omega_1, \omega_2, \dots, \omega_{N_\omega}\} \sim \mathcal{D}_\omega$.
 - 16 Calculate $d\theta_\pi$ and $d\theta_v$ using Eq. (14)(15)
 - 17 Perform asynchronous update of θ_v using $d\theta_v$ and of θ_π using $d\theta_\pi$.
 - 18 **return** θ_π and θ_v ...
-

3.3.3 Training algorithm

Combine the above components, we get the agent training algorithm as shown in Algorithm 1, which works as follows. We first initialize the replay buffer as an empty set (line 1) and adopt a uniform distribution for preference sampling (line 2). At the beginning of each episode, the agent synchronizes the network parameters with the global parameters (line 4). Next, it interacts with the environment to collect the trajectory for the replay buffer (lines 5-12). Considering the needly-in-the-haystack problem, the step length is calculated based on the early termination trick (lines 7-8). Then the stochastic gradient of the parameters for the actor network ($d\theta_\pi$) and for the critic network ($d\theta_v$) are calculated according to the sampled transitions and the homotopy update trick (lines 13-

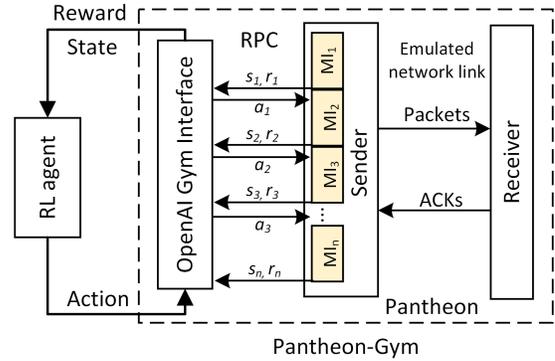


Figure 3: Pantheon-Gym architecture.

16). Finally, the global parameters are updated asynchronously based on $d\theta_\pi$ and $d\theta_v$ (line 17).

Essentially, Algorithm 1 is to iteratively apply the Bellman optimality operator \mathcal{T} on \mathbf{Q} . Since \mathcal{T} is a contraction mapping on the complete pseudo-metric space, Algorithm 1 will finally terminate with a MOQ-value function \mathbf{Q} that is equivalent to the preferred optimal value function \mathbf{Q}^* based on the Multi-Objective Banach Fixed-Point Theorem [30].

3.4 Pantheon-Gym: MORL Training Suite For CC

To train the models and to facilitate further research, we present Pantheon-Gym, which is an asynchronous MORL suite for CC based on the OpenAI Gym interface [32] and Pantheon [7] as shown in Figure 3. It supports MORL by returning a reward vector rather than a single scalar. Meanwhile, it interacts with the MORL agent with the standard OpenAI Gym interface, letting researchers design their own agent in an easy way.

Besides the features adopted in our state space, it also supports various kinds of features including 1) sending rate; 2) receiving rate; 3) 95th percentile latency; 4) the exponentially-weighted moving average (EWMA) of the queuing delay; 5) the EWMA of the sending rate; 6) the EWMA of the receiving rate. Users can conveniently customize their own reward functions and state space.

Compared with Aurora [5] which proposes a training suite based on a simulated network environment, Pantheon-Gym provides more realistic training data by adopting an emulated network environment. To promote bandwidth utilization and training efficiency, Pantheon-Gym trains models in an asynchronous fashion similar to MVFST-RL [11]. The agents, Gym interface and the emulated network work in different processes and communicate with each other through remote procedure calls. Therefore, the environment steps are not blocked by the policy lookup and the forward-pass in the training stage.

4 Environment-Adaptive Preference Selection Method

In this section, we propose a novel environment-adaptive preference selection method by training the preference adaptation model with the help of the expert policy. When the preference is not determined by the upper-level application, it automatically selects a suitable preference by taking the state sequences as input to recognize the network environment.

4.1 Expert Policy

In the training stage, since we know the link capacity, we can construct an expert policy π^* that adjusts the sending rate to the link capacity as quickly as possible. More specifically, assuming c is the link capacity, then the action produced by the expert policy at each MI is depicted as

$$\pi^* : a_t = \arg_{a \in \mathcal{A}} \min(|\text{apply}(sr_t, a) - c| + c * \mathbb{1}_{\text{apply}(sr_t, a) > c}) \quad (18)$$

where $\mathbb{1}_{\text{apply}(sr_t, a) > c}$ is a binary function that outputs 1 if $\text{apply}(sr_t, a)$ is larger than the link capacity c . It guarantees that actions that do not cause excessive sending rate will be prioritized.

4.2 Policy Similarity Model

We next propose a policy similarity model based on the cumulative reward distributions [33], which is used to quantify the similarity between the expert policy and the policy for a given preference ω . Based on it, we find the best preference for each training environment. Compared with ordinary methods that only use the expectations to estimate the policy, such a method achieves higher accuracy.

The cumulative reward for policy π is defined as

$$\hat{\mathcal{R}}^\pi = \sum_{t=0}^T \gamma^t \mathcal{R}(s_t, a_t). \quad (19)$$

Due to the transition probability, $\hat{\mathcal{R}}^\pi$ is a random variable in \mathbb{R}^2 . For a given policy π , we further define its Markov chain as $\mathcal{M}(\pi) = \{\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \pi\}$. Then the cumulative distribution of $\hat{\mathcal{R}}^\pi$ can be formulated as,

$$P^\pi(\boldsymbol{\epsilon} | s, a) = \Pr(\hat{\mathcal{R}}^\pi \leq \boldsymbol{\epsilon} | s, a, \mathcal{M}(\pi)) \quad (20)$$

By assuming this distribution follows a multi-variable Gaussian distribution $G(\boldsymbol{\epsilon}; \theta)$ parameterized by θ , and adopting a Monte Carlo sampling technique to collect dataset \mathcal{D} , we can get the distribution parameters by,

$$\Theta^* = \arg_{\Theta} \max \mathbb{E}_{\mathcal{D}}(G(\boldsymbol{\epsilon}; \Theta)) \quad (21)$$

We then use the Kullback-Leibler divergence [34] between the expert and the policy generate by different preference as the similarity measurement, which is formulated by

$$D_{\text{KL}}(G^* \| G_\omega) = \int_{-\infty}^{\infty} G(\boldsymbol{\epsilon}; \Theta^*) \log \left(\frac{G(\boldsymbol{\epsilon}; \Theta^*)}{G(\boldsymbol{\epsilon}; \Theta_\omega)} \right) d\boldsymbol{\epsilon} \quad (22)$$

where $G^* = G(\boldsymbol{\epsilon}; \Theta^*)$ and $G_\omega = G(\boldsymbol{\epsilon}; \Theta_\omega)$ are the Gaussian distributions for the expert policy and the policy generated by preference ω . A larger D_{KL} means that the similarity between the two policies is lower.

4.3 Preference Adaptation Model

Based on the policy similarity model, we first find the most suitable preference ω for each training environment, under which the reward distribution is most similar to that of the expert policy (i.e., $D_{\text{KL}}(G^* \| G_\omega)$ is minimized). Then, by running a set of experiments using an indicator preference $\tilde{\omega}$, we can collect the training dataset $\mathcal{D}_p = \{(\mathcal{S}_0, \omega_0^*), (\mathcal{S}_1, \omega_1^*), \dots, (\mathcal{S}_n, \omega_n^*)\}$, where \mathcal{S}_i is a state trajectory, and ω_i^* is the most suitable preference for \mathcal{S}_i .

Build upon this dataset, the policy adaptation problem can be formulate as a classification problem where each candidate preference can be seen as a class. To solve this problem, we train a *preference adaptation model* named *AdaM* using supervised learning. *AdaM* is a three-layer neural network. The first two layers of it are Long Short-Term Memory (LSTM [35]) layers that can efficiently handle the sequence data while the last layer is a fully connected layer.

We adopt the cross entropy loss [36] to train *AdaM*, which is formulated as

$$\mathcal{L} = - \sum_{i=0}^N \omega_i \log(\omega_i^*) \quad (23)$$

where ω_i^* is the ground truth for training sample i , while ω_i is the prediction. We omit the details of the training procedure since it is a classical supervised learning.

Algorithm 2: Preference Adaptation Algorithm

Input: Trained model *AdaM* and π ; indicator preference $\tilde{\omega}$; sample count SC .

Output: Optimal ω for the current environment.

- 1 Initialize sequence buffer $\mathcal{S} = \{\}$.
 - 2 Observe the current state s according to Eq. (4).
 - 3 Get instruction policy $\pi_{\tilde{\omega}}$
 - 4 **for** k from 1 to SC **do**
 - 5 Sample an action $a_t \sim \pi_{\tilde{\omega}}(s)$.
 - 6 Observe a new state s' .
 - 7 Set $\mathcal{S} = \mathcal{S} \cup s'$.
 - 8 Set $s = s'$.
 - 9 **return** $\omega = \text{AdaM}(\mathcal{S})$.
-

Once we train the *AdaM*, we can use Algorithm 2 to infer the most suitable preference. It first initializes the state sequence buffer and observes the initial state (lines 1-2). Then, it collects a state sequence by interacting with the environment using the policy generated by the indicator preference $\tilde{\omega}$ (lines 3-8). Finally, we infer the best preference using the trained adaptation model *AdaM* (line 9).

5 Implementation and Evaluations

5.1 System Settings

Model Training and AUTO Implementation. We adopted the same training parameters as Aurora [5] to train the MORL agent as shown in Table 1. All parameters are sampled uniformly except the queue size, for which the log is sampled uniformly. The history length is set to 10. To train the preference adaptation model, we set the sample count SC to 10 and set the indicator preference $\tilde{\omega}$ to $[0.3, 0.7]$, i.e., the weights of throughput and delay are 0.3 and 0.7 respectively.

Table 1: Training Parameters

Bandwidth	Latency	Queue size	Loss rate
100-500 pps	50-500 ms	2-2981 packets	0-5%

We train models with 16 workers in parallel with different sampled preferences. It takes around 7 days on a server with Intel Xeon Gold 6148 CPU and 128G memory to converge. The whole model takes about 0.5 milliseconds on our server. Using the two trained models, we implemented a user-space prototype on top of Pytorch [37] and the UDT framework [38]. It only requires modifications on the sender side and it works smoothly when any other CC scheme on the receiver-side sends per-packet ACK.

Evaluation environments. We evaluate performances in both emulated environments and the real Internet through following tools.

- Pantheon [7]. The Pantheon is a community evaluation platform. It supports performance evaluations on the Internet and meanwhile can generate *calibrated network emulators* that capture the diverse performance of real Internet paths. It only supports point-to-point topology and can run one scheme at once.
- CoCo-Beholder [39]. The CoCo-Beholder is an emulator based on Pantheon. It emulates multiple flows with different CC schemes on a dumbbell topology.

Compared CC algorithms. We compare AUTO with state-of-the-art methods that have been proven to achieve high performance in the heterogeneous network, including heuristic-based CC algorithms FillP [40], Copa [4], BBR [41] and LEDBAT [42], online learning based method PCC-Allegro [16] and Vivace [17], inverse RL based method Indigo [7], RL-based method Aurora [5], and statistic-based method TaoVA [43]. We further adopt classic TCP variants including TCP Cubic [9] and TCP Vegas [44] as baselines.

5.2 Achieve Consistent High Performance in Different Environments

We first show AUTO achieves consistent high performance in different network environments by evaluating its perfor-

mance in three real-world, representative, and very different environments and a hybrid network environment.

5.2.1 Satellite network

The first representative environment is the satellite network, where links typically have longer RTT. We evaluate the performance on Pantheon parameterized with the real-world measurements of the WINDs satellite system [6], replicating an experiment from the PCC-Allegro paper [16]. The link has 800 ms RTT, 42 Mbps capacity, 0.74% stochastic loss rate, and a queue size of 1500. Part of the comparison algorithms have been shown to achieve high performance in this environment, including PCC-Allegro [16], Vivace [17] and Copa [4].

Figure 4(a) shows as expected, AUTO achieves different trade-off results between delay and throughput by inputting different preferences. Therefore, AUTO can cope with different application requirements. In contrast, other methods only obtain a single trade-off result.

For each comparison algorithm, there is always a preference under which AUTO achieves a better performance. Compared with methods that work well in satellite network environments, AUTO (0.8, 0.2)¹ achieves similar throughput but 14% shorter delay compared to PCC-Allegro, and 30.0% higher throughput and 20.6% shorter delay than Vivace, while AUTO (0.5, 0.5) achieves 6.9% higher throughput and 12.2% shorter delay than Copa. Compared with Indigo, the throughput and the delay of AUTO (0.2, 0.8) are 19.9% higher and 5.0% shorter respectively. Compared with BBR, AUTO (0.2, 0.8) achieves 153.5% higher throughput and 5.7% shorter delay. Compared with schemes that achieve short delay, the throughput of AUTO (0.05, 0.95) is about 1.1 times of FillP, 3.1 times of Aurora, 4.0 times of TaoVA, 15.3 times of TCP Cubic, 29.2 times of TCP Vegas, and 86 times of LEDBAT.

In this environment, (0.5, 0.5) is selected as the default preference by the preference adaptation model. Its corresponding packet loss rate is shown in the blue bar in Figure 5, which is 79.0% and 85.9% lower than Copa and PCC-Allegro that achieve comparable throughput. With much higher throughput, the packet loss rate of AUTO (0.3, 0.7) is also lower than FillP, BBR, Vivace, and Aurora.

5.2.2 Cellular network

The second representative environment is the cellular network, where the links have time-varying speeds and are usually modeled as Poisson Point Processes (PPP). We emulate this environment on Pantheon parameterized with the real-world measurements of the path from Amazon Web Services (AWS) in Brazil to the Colombia cellular network, replicating an experiment from Pantheon [7]. The link has 260 ms RTT, 3.04 Mbps capacity, 0.6% stochastic loss rate, and a queue size of 426.

With different preferences, AUTO outperforms other meth-

¹We use AUTO (ω_t, ω_d) to represent AUTO with preference set to $[\omega_t, \omega_d]$.

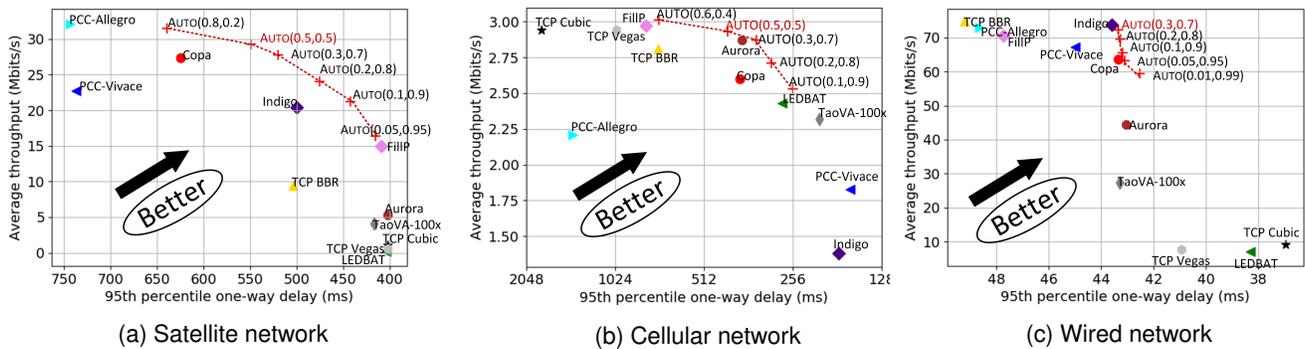


Figure 4: Throughput v.s. 95th percentile one-way delay in three representative environments. AUTO achieves consistent high performance and can cope with different application requirements by achieving a series the Pareto dominate results.

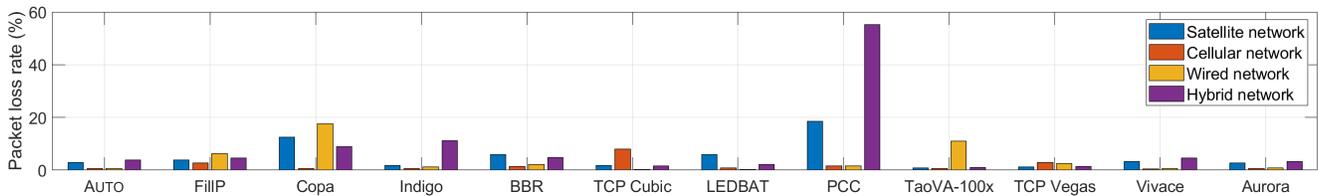


Figure 5: Packet loss rate in the satellite, cellular, wired, and hybrid network environments

ods in different aspects as shown in Figure 4(b). By setting the preference to (0.6, 0.4), AUTO achieves the highest throughput. Compared with other methods that prefer high throughput, it achieves 61.1%, 28.2% and 9.7% shorter delay than TCP Cubic, TCP Vegas and FillIP respectively. By putting more weight on delay, AUTO achieves shorter delay at the expense of lower throughput. Statistically, AUTO (0.3, 0.7) achieves 5.7% higher throughput and 16.3% shorter delay than Copa. With almost the same throughput, it achieves 10.2% lower latency compared with Aurora. AUTO (0.1, 0.9) achieves 4.1% higher throughput and 8.3% shorter delay than LEDBAT. For other methods that achieve low link utilization in this environment, AUTO (0.1, 0.9) achieves 9.1%, 14.5%, 38.3% and 83.3% higher throughput than TaoVA, PCC-Allegro, Vivace and Indigo respectively.

The default preference in this environment is (0.5, 0.5). Under this preference, the packet loss rate of AUTO is 0.64% and is lower than all other methods except Vivace, which achieves 0.56% packet loss rate but very low link utilization.

5.2.3 Wired network

The third representative environment is the wired network, where the links have higher bandwidth and shorter buffer. We evaluate the performance in this environment on the real Internet by utilizing two servers located at Shanghai and Hong Kong respectively. The link between the two server has about 36.2 ms RTT and 80 Mbps capacity.

Although the link bandwidth in the evaluate environment is far beyond the training range [1.12 Mbps, 5.6 Mbps], AUTO

still achieves a set of Pareto dominant results as shown in Figure 4(c). This is because the adopted state space achieves a good generalization of the model. By setting the preference as FillIP, Copa, PCC-Allegro, Vivace and Indigo and meanwhile achieves shorter delay. Due to the stochastic packet loss, TCP Cubic, TCP Vegas and LEDBAT have poor performance in this test. Due to the slow convergence, the throughput of Aurora is 38.5% lower than AUTO (0.3, 0.7).

In this test, the queuing delay fluctuate in a small range because of the small queue size. If setting the throughput weight too large (≥ 0.4), AUTO will neglect the small queuing delay, which will result in high packet loss rate. Therefore, we set the throughput weight no higher than 0.3. In contrast, the delay-based method Copa cannot detect congestion correctly and therefore has large packet loss rate as shown in Figure 5.

Similar to the previous results, the default preference setting (0.3, 0.7) achieves lower packet loss rate than other CC algorithms that achieve the similar throughput. Quantitatively, the packet loss rate of AUTO (0.3, 0.7) is 88.9% lower than FillIP, 96.0% lower than Copa, 42.1% lower than Indigo, 57.3% lower than PCC-Allegro and is 2.7% lower than Vivace.

5.2.4 Hybrid network

To show AUTO can cope with the heterogeneous satellite-ground integrated network, we next evaluate the performance on an emulated hybrid network path using Pantheon. The hybrid path contains a satellite link with 800 ms RTT and 42 Mbps bandwidth, and a terrestrial link with 40 ms RTT and

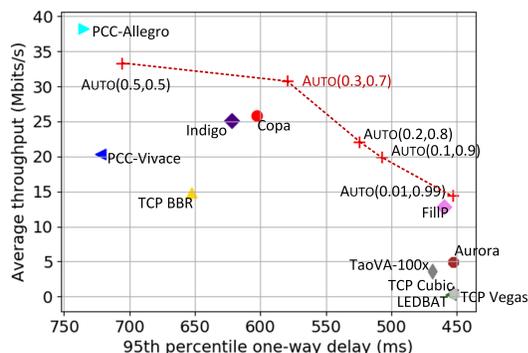


Figure 6: Throughput v.s. 95th percentile one-way delay in the hybrid network environment.

60 Mbps bandwidth.

For congestion control, the hybrid path can be abstracted as a single special link. Its delay and packet loss rate are the total delay and the overall packet loss rate of the path respectively, while its bandwidth is the bandwidth of the bottleneck link on the path. Since the bandwidth-delay ratios of hybrid paths are covered by representative environments, AUTO can well cope with it. As shown in Figure 6, AUTO achieves both high throughput and low latency. With similar latency, AUTO(0.01, 0.99) achieves 14.36 Mbps throughput, corresponding to a gain of 52.2 \times , 30.9 \times , 18.9 \times , 2.9 \times , 1.9 \times , and 12.5% over LEDBAT, TCP Vegas, TCP Cubic, TaoVA, Aurora, and FillP respectively. With similar throughput, it also improves the delay by 30.7% compared with BBR. AUTO(0.1, 0.9) achieves similar delay with Vivace but its delay (507.2 ms) is 29.8% shorter. Compared with Copa, AUTO(0.3, 0.7) improves the throughput and delay by 18.6% and 3.9% respectively. Meanwhile, it achieves 22.5% higher throughput and 6.9% lower delay compared with Indigo.

From the perspective of packet loss rate, the default preference setting (0.3, 0.7) performs better than all other methods that achieve a link utilization rate higher than 15%. As shown in Figure 5, it still achieves 15.0%, 16.9%, 18.9%, 57.0%, 65.5% lower packet loss rate than FillP, Vivace, BBR, Copa, and Indigo respectively. Although PCC-Allegro achieves the highest throughput in this scenario, it fails to detect congestion since the chosen preference is not suitable for this scenario. It suffers from 55.11% packet loss rate, which is 13.3 \times higher than AUTO(0.3, 0.7).

5.2.5 Summary

Existing CC methods cannot adapt to diversified environments. For example, BBR and FillP suffer from low throughput or long delay in the satellite, cellular, and the hybrid network environments. Due to the fixed monitoring interval and lack of consideration on delayed action phenomenon, Indigo can only achieve high performance in the wired network environment. Due to the empirical preference settings, PCC

suffers from unacceptably high packet loss in the satellite environment, Vivace suffers from low throughput in both satellite and cellular network environments, while Aurora has poor performance in the satellite and the wired network environments.

In contrast, AUTO adapts to different network environments by automatically adjusting the preference. Although we adopt a relatively small training range, the trained policy model achieve high performance since there is no absolute value in the state space. Its performance can be further improved by adding more training environments. Further, by finding the Pareto-optimal frontier, AUTO can cope with different application requirements.

5.3 Adapt to Rapid Network Changes

We next show AUTO achieves high reactivity and adapts to the rapid network changes by emulating a link with dynamic available bandwidth, which changes every 5 seconds with a uniform distribution ranging from 2 Mbps to 10 Mbps. The link delay and loss rate are set to 50 ms and 1% respectively.

Figure 7(a) shows that AUTO achieves highest average throughput. Quantitatively, it reaches 5.95 Mbps and 92.3% link utilization, corresponding to a gain of 2.6%, 5.1%, 7.0%, 11.8%, 16.9%, 37.4%, 45.5%, 75%, 1.58 \times , 2.79 \times , 3.92 \times over FillP, Vivace, BBR, Copa, TaoVA, Indigo, PCC-Allegro, Aurora, LEDBAT, TCP Vegas and Cubic, respectively. To further demonstrate AUTO's reactivity, we show AUTO achieves shorter delay comparing to other algorithms that achieve comparable throughput in the blue bar. More specifically, compared with FillP, Vivace and BBR, the delay of AUTO is 25.4%, 96.6%, and 74.7% shorter.

Figures 7(b)-(m) illustrate the behavior of different CC methods across time. As shown in Figure 7(b), AUTO's sending rate fluctuates around the available bandwidth since it adjusts the sending rate mainly according to delay variation. It tries to increase the sending rate when the latency ratio is small and to decrease otherwise. FillP and BBR also achieve good performance but have longer converge time when the bandwidth suddenly increases. Although Vivace achieves high throughput, it cannot cope with the bandwidth decrease and result in extremely long delay and high packet loss rate. Due to the slow convergence, Aurora has low reactivity and suffers from poor bandwidth utilization. Other algorithms such as Copa, Indigo, Cubic, Vegas, and LEDBAT cannot cope with the variant bandwidth or misestimate the network status due to the stochastic packet loss rate. Their and suffer from low throughput.

5.4 Resilient to Stochastic Packet Loss

Then, we show that AUTO is resilient to stochastic packet loss, which is challenging for CC algorithms since the stochastic packet loss caused by poor link quality could let CC algorithms mistakenly think that congestion occurred and further

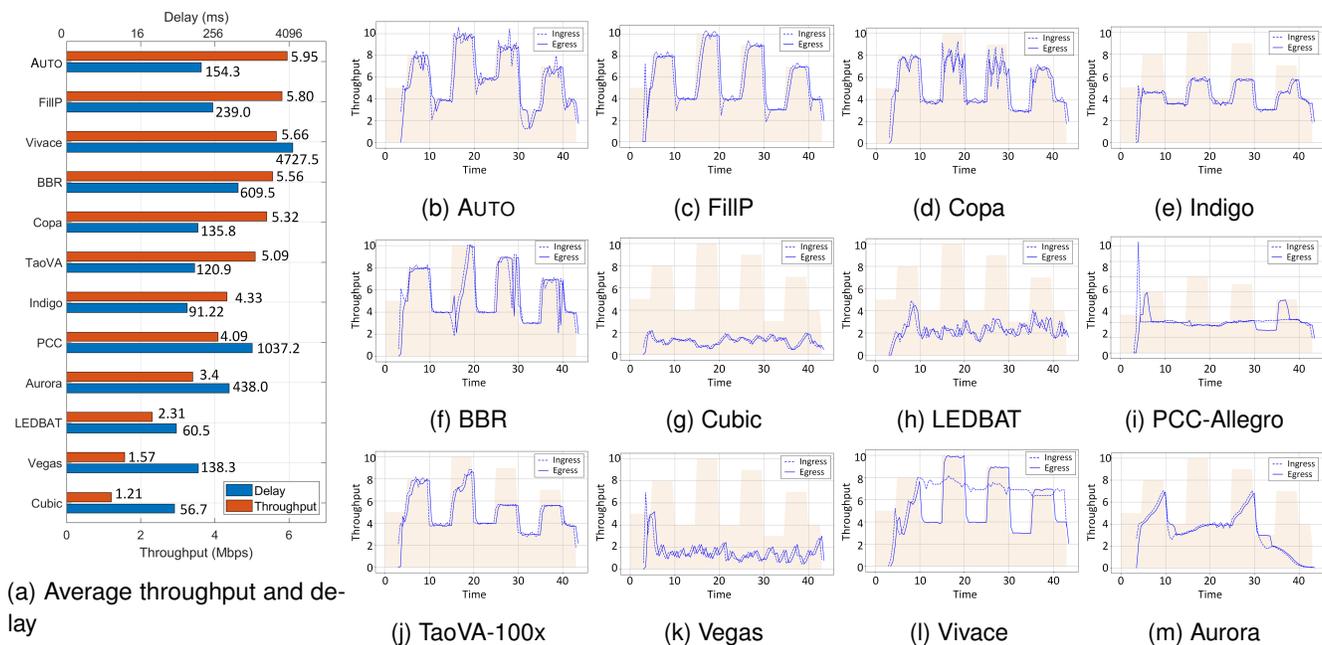


Figure 7: Performance evaluations in the rapidly changing network scenario.

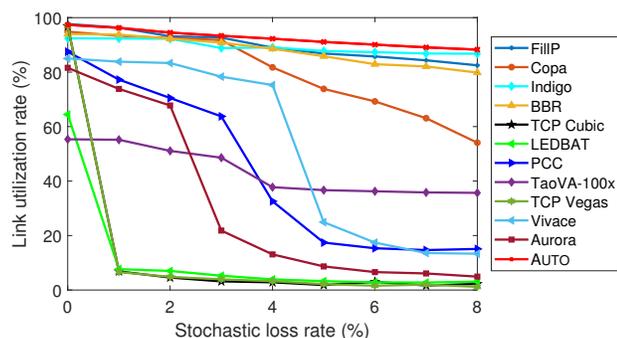


Figure 8: Link utilization v.s. stochastic loss rate.

decrease the sending rate. We test AUTO against other algorithms under an emulated link with a rate of 12 Mbps bandwidth and an RTT of 200 ms. To further test the robustness of the trained policy model, we let the stochastic loss rate of the link range from 0% to 8%, which is beyond our training range [0%, 5%].

As shown in Figure 8, AUTO achieves consistent high link utilization under all stochastic loss rates. Even when the stochastic loss rate is set to 8%, which is outside the training range, AUTO achieves 88.25% link utilization, which is 6.9% higher than the second-highest method FillP. The main reason is that AUTO judges whether congestion occurs by jointly considering the latency ratio, the latency gradient, the sending ratio, and the loss rate. Since the stochastic loss rate only influences the last two features, AUTO still can detect congestion accurately. With a similar reason, Indigo also achieves high

link utilization. In contrast, the link utilization of Aurora falls sharply when the stochastic loss rate larger than 2% due to the fixed preference. BBR adjusts the sending rate according to the estimated bandwidth and minimum latency, thus, it is less affected by the stochastic loss rate. Since packet loss is not considered in the packet arrival model, the performance of Copa degrades when the stochastic loss rate is larger than 3%. TaoVA fails to consider lots of real network parameters such as stochastic loss rate and link buffer size and its link utilization ranges from 35% to 55%. For PCC-Allegro and Vivace that adopt the fixed weight on the packet loss rate in the utility function, their performance is poor when the stochastic packet loss rate is large since the calculated utilities are always low. As for TCP Cubic, TCP Vegas and LEDBAT, they are designed for reliable network environments and are sensitive to packet loss. They achieve relatively high link utilization when there is no stochastic packet drop on the link, and their link utilization drops to about 3% when the stochastic loss rate increases to 8%.

5.5 Fairness Against Different CC Schemes

Last, we show that AUTO is competitive-configurable and can achieve fairness against different CC schemes with different competitiveness, which solves the pain point of existing offline learning-based CC methods [5]. We first set up two flows on a dumbbell topology using CoCo-Beholder [39] with one flow using AUTO and competing with another flow using other CC schemes. The bottleneck link has 100 ms RTT, 20 Mbps capacity, and a queue size of 200. After 30 seconds running, we evaluate the Jain's Fairness Index (JFI) of the

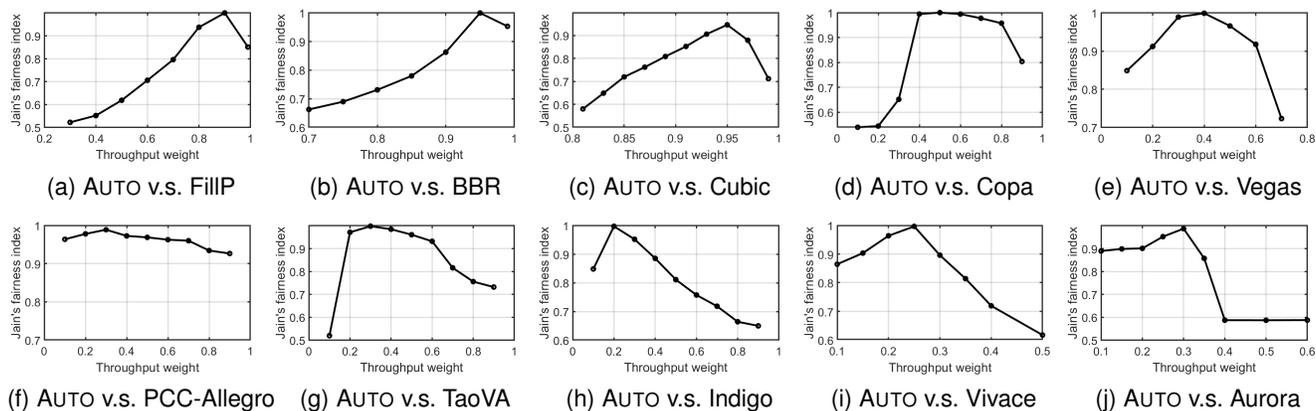


Figure 9: JFI between AUTO and other CC schemes v.s. the throughput weight (ω_t). The delay weight is set to $1 - \omega_t$.

two flows, which is calculated by \bar{x}^2/\bar{x}^2 and $\mathbf{x} = [x_{\text{AUTO}}, x_C]$ is the throughput of AUTO and the competing scheme respectively. The JFI ranges from 0.5 (worst case, the throughput of one flow is 0) to 1 (best case, the two flows have the same throughput).

As shown in Figure 9, the JFI first increases with the throughput weight and then decreases. This is because the competitiveness of AUTO increases along with the throughput weight so that the JFI increases before AUTO achieves the same competitiveness as the competing flow and decreases after. More specifically, with a small throughput weight, AUTO is extremely delay-sensitive and tends to decrease the sending rate when the competing flow causes the increase of the queuing delay. On the contrary, if higher weight is put on throughput, AUTO will be more competitive and decrease the sending rate only if the latency ratio is large.

Therefore, AUTO can always find a suitable preference such that the two flows achieve comparable throughput on the shared link. When competing with CC schemes with high competitiveness, AUTO achieves the fairness by increasing the throughput weight. For example, FillP, BBR and TCP Cubic have high competitiveness for the first two adjust the sending rate based on the estimated bandwidth and the last reduces the congestion window size only when there is packet loss. As shown in Figures 9(a)-(c), by setting preference to (0.9, 0.1), (0.95, 0.05) and (0.95, 0.05), AUTO achieves the fairness against FillP, BBR and TCP Cubic respectively. In contrast, by putting higher weight on delay, AUTO can play well with delay-sensitive CC algorithms. Among the selected CC schemes, Indigo, Vivace and Aurora are more delay-sensitive due to their selected features or the rate adjusting mechanism. As shown in Figure 9(h)-(j), AUTO achieves the fairness against these three algorithms by setting the preference to (0.2, 0.8), (0.25, 0.75), and (0.3, 0.7) respectively.

Meanwhile, the competitive-configurable characteristic also supports users to explicitly set the priority for different flows. For example, users can adjust the throughput weight

of flows that require high bandwidth (e.g., file downloading) to larger than that of delay-sensitive flows (e.g., online meeting). By doing so, the performance of delay-sensitive can be guaranteed while other flows only use the residue bandwidth.

6 Conclusion

We proposed AUTO, an adaptive CC algorithm based on multi-objective reinforcement learning for the heterogeneous satellite-ground integrated network. It doesn't target at finding the optimal result for a single preference, but instead aims for finding *optimal policies for all possible preferences*. Thus, AUTO can cope with heterogeneous network environments and diversified application requirements using a single agent, by taking different preferences as input. To adjust the preference adaptively in different environments, we next proposed an environment-adaptive preference selection method by training a preference adaptation model. To train the agent on emulated environments and to facilitate further research, we developed a training suite Pantheon-Gym for MORL-based CC. Evaluation results show that AUTO achieves consistent high performance in representative network environments, adapts to rapid network changes, is resilient to stochastic packet loss, and can achieve fairness against different CC schemes.

Acknowledgments

We would like to thank the anonymous reviewers for their thoughtful comments. This work was supported in part by the National Natural Science Foundation of China projects (Nos. 61832013), in part by STCSM (Science and Technology Commission of Shanghai Municipality) AI project (No.19511120300), in part by the National Key Research and Development Program of China (No. 2019YFB2102204), and in part by the Huawei Technologies Co., Ltd projects (Nos. YBN2019105155 and YBN2020085026). Feilong Tang is the corresponding author of this paper.

References

- [1] J. Liu, Y. Shi, Z. M. Fadlullah, and N. Kato, "Space-air-ground integrated network: A survey," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 2714–2741, 2018.
- [2] F. Tang, "Dynamically adaptive cooperation transmission among satellite-ground integrated networks," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 1559–1568.
- [3] M. Polese, F. Chiariotti, E. Bonetto, F. Rigotto, A. Zanella, and M. Zorzi, "A survey on recent advances in transport layer protocols," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3584–3608, 2019.
- [4] V. Arun and H. Balakrishnan, "Copa: Practical delay-based congestion control for the internet," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 329–342.
- [5] N. Jay, N. H. Rotman, P. Godfrey, M. Schapira, and A. Tamar, "Internet congestion control via deep reinforcement learning," *arXiv preprint arXiv:1810.03259*, 2018.
- [6] H. Obata, K. Tamehiro, and K. Ishida, "Experimental evaluation of tcp-star for satellite internet over winds," in *Proceedings of the International Symposium on Autonomous Decentralized Systems (ISADS)*. IEEE, 2011, pp. 605–610.
- [7] F. Y. Yan, J. Ma, G. D. Hill, D. Raghavan, R. S. Wahby, P. Levis, and K. Winstein, "Pantheon: the training ground for internet congestion-control research," in *USENIX Annual Technical Conference (ATC)*, 2018, pp. 731–743.
- [8] S. Abbasloo, C.-Y. Yen, and H. J. Chao, "Classic meets modern: a pragmatic learning-based congestion control for the internet," 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*, 2020, pp. 632–647.
- [9] S. Ha, I. Rhee, and L. Xu, "Cubic: a new tcp-friendly high-speed tcp variant," *ACM SIGOPS operating systems review*, vol. 42, no. 5, pp. 64–74, 2008.
- [10] R. Al-Saadi, G. Armitage, J. But, and P. Branch, "A survey of delay-based and hybrid tcp congestion control algorithms," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3609–3638, 2019.
- [11] V. Sivakumar, T. Rocktäschel, A. H. Miller, H. Küttler, N. Nardelli, M. Rabbat, J. Pineau, and S. Riedel, "Mvfst-rl: An asynchronous rl framework for congestion control with delayed actions," *arXiv preprint arXiv:1910.04054*, 2019.
- [12] Y. Kong, H. Zang, and X. Ma, "Improving tcp congestion control with machine intelligence," in *Proceedings of the 2018 Workshop on Network Meets AI & ML*, 2018, pp. 60–66.
- [13] W. Li, F. Zhou, K. R. Chowdhury, and W. Meleis, "Qtcp: Adaptive congestion control with reinforcement learning," *IEEE Transactions on Network Science and Engineering*, vol. 6, no. 3, pp. 445–458, 2018.
- [14] H. Mao, P. Negi, A. Narayan, H. Wang, J. Yang, H. Wang, R. Marcus, M. K. Shirkoobi, S. He, V. Nathan *et al.*, "Park: An open platform for learning-augmented computer systems," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019, pp. 2490–2502.
- [15] X. Nie, Y. Zhao, Z. Li, G. Chen, K. Sui, J. Zhang, Z. Ye, and D. Pei, "Dynamic tcp initial windows and congestion control schemes through reinforcement learning," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1231–1247, 2019.
- [16] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "Pcc: Re-architecting congestion control for consistent high performance," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015, pp. 395–408.
- [17] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira, "Pcc vivace: Online-learning congestion control," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 343–356.
- [18] T. Meng, N. R. Schiff, P. B. Godfrey, and M. Schapira, "Pcc proteus: Scavenger transport and beyond," 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*, 2020, pp. 615–631.
- [19] K. Winstein and H. Balakrishnan, "Tcp ex machina: Computer-generated congestion control," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 123–134, 2013.
- [20] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.
- [21] T. Issariyakul and E. Hossain, "Introduction to network simulator 2 (ns2)," in *Introduction to network simulator NS2*. Springer, 2009, pp. 1–18.

- [22] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, "Network simulations with the ns-3 simulator," *SIGCOMM demonstration*, vol. 14, no. 14, p. 527, 2008.
- [23] W. Li, H. Zhang, S. Gao, C. Xue, X. Wang, and S. Lu, "Smartcc: A reinforcement learning approach for multi-path tcp congestion control in heterogeneous networks," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 11, pp. 2621–2633, 2019.
- [24] Z. Xu, J. Tang, C. Yin, Y. Wang, and G. Xue, "Experience-driven congestion control: When multi-path tcp meets deep reinforcement learning," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1325–1336, 2019.
- [25] F. Ruffy, M. Przystupa, and I. Beschastnikh, "Iroko: A framework to prototype reinforcement learning for data center traffic control," *arXiv preprint arXiv:1812.09975*, 2018.
- [26] T. L. Hayes, K. Kafle, R. Shrestha, M. Acharya, and C. Kanan, "Remind your neural network to prevent catastrophic forgetting," in *European Conference on Computer Vision*. Springer, 2020, pp. 466–483.
- [27] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proceedings of the International conference on machine learning (ICML)*, 2016.
- [28] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," *IEEE Transactions on Neural Networks*, vol. 16, pp. 285–286, 2005.
- [29] j. schulman, p. moritz, s. levine, I. M. Jordan, and p. abbeel, "High-dimensional continuous control using generalized advantage estimation," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.
- [30] R. Yang, X. Sun, and K. Narasimhan, "A generalized algorithm for multi-objective reinforcement learning and policy adaptation," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019, pp. 14 610–14 621.
- [31] T. L. Watson and T. R. Haftka, "Modern homotopy methods in optimization," *Computer Methods in Applied Mechanics and Engineering*, pp. 289–305, 1989.
- [32] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.
- [33] T. Morimura, M. Sugiyama, H. Kashima, H. Hachiya, and T. Tanaka, "Nonparametric return distribution approximation for reinforcement learning," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2010.
- [34] S. Kullback and R. A. Leibler, "On information and sufficiency," *Annals of Mathematical Statistics*, vol. 22, pp. 79–86, 1951.
- [35] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735–1780, 1997.
- [36] J. Shore and R. P. Johnson, "Properties of cross-entropy minimization," *IEEE Trans. Inf. Theory*, vol. 27, pp. 472–482, 1981.
- [37] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.
- [38] Y. Gu, *UDT: a high performance data transport protocol*. University of Illinois at Chicago, 2005.
- [39] E. Khasina, "The coco-beholder: Enabling comprehensive evaluation of congestion control algorithms," *arXiv preprint arXiv:1912.10531*, 2019.
- [40] T. Li, K. Zheng, K. Xu, R. A. Jadhav, T. Xiong, K. Winstein, and K. Tan, "Tack: Improving wireless transport performance by taming acknowledgments," in *Proceedings of the ACM SIGCOMM Conference*, 2020, pp. 15–30.
- [41] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "Bbr: Congestion-based congestion control," *Queue*, vol. 14, no. 5, pp. 20–53, 2016.
- [42] D. Rossi, C. Testa, S. Valenti, and L. Muscariello, "Ledbat: the new bittorrent congestion control protocol," in *Proceedings of the International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2010, pp. 1–6.
- [43] A. Sivaraman, K. Winstein, P. Thaker, and H. Balakrishnan, "An experimental study of the learnability of congestion control," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 479–490, 2014.
- [44] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "Tcp vegas: New techniques for congestion detection and avoidance," in *Proceedings of the ACM SIGCOMM Conference*, 1994, pp. 24–35.

Hey, Lumi! Using Natural Language for Intent-Based Network Management

Arthur S. Jacobs
UFRGS

Ricardo J. Pfitscher
UFRGS

Rafael H. Ribeiro
UFRGS

Ronaldo A. Ferreira
UFMS

Lisandro Z. Granville
UFRGS

Walter Willinger
NIKSUN, Inc.

Sanjay G. Rao
Purdue University

Abstract

In this work, we ask: what would it take for, say, a campus network operator to tell the network, using natural language, to “*Inspect traffic for the dorm*”? How could the network instantly and correctly translate the request into low-level configuration commands and deploy them in the network to accomplish the job it was “asked” to do? We answer these questions by presenting the design and implementation of LUMI, a new system that (i) allows operators to express intents in natural language, (ii) uses machine learning and operator feedback to ensure that the translated intents conform with the operator’s goals, and (iii) compiles and deploys them correctly in the network. As part of LUMI, we rely on an abstraction layer between natural language intents and network configuration commands referred to as *Nile* (Network Intent Language). We evaluate LUMI using synthetic and real campus network policies and show that LUMI extracts entities with high precision and compiles intents in a few milliseconds. We also report on a user study where 88.5% of participants state they would rather use LUMI exclusively or in conjunction with configuration commands.

1 Introduction

Deploying policies in modern enterprise networks poses significant challenges for today’s network operators. Since policies typically describe high-level goals or business intents, the operators must perform the complex and error-prone job of breaking each policy down into low-level tasks and deploying them in the physical or virtual devices of interest across the entire network. Recently, *intent-based networking (IBN)* has been proposed to solve this problem by allowing operators to specify high-level policies that express how the network should behave (e.g., defining goals for quality of service, security, and performance) without having to worry about how the network is programmed to achieve the desired goals [17]. Ideally, IBN should enable an operator to simply tell the network to, for example, “*Inspect traffic for the dorm*”, with the network instantly and correctly breaking down such an intent into configurations and deploying them in the network.

In its current form, IBN has not yet delivered on its promise of fast, automated, and reliable policy deployment. One of the main reasons for this shortcoming is that, while network policies are generally documented in natural language, we cannot currently use them as input to intent-based management systems. Despite growing interest from some of the largest tech companies [7, 47, 62] and service providers [32, 38, 52], only a few research efforts [4, 13] have exploited the use of natural language to interact with the network, but they lack support for IBN or other crucial features (e.g., operator confirmation and feedback). However, expressing intents directly in natural language has numerous benefits when it comes to network policy deployment. For one, it avoids the many pitfalls of traditional policy deployment approaches, such as being forced to learn new network programming languages and vendor-specific Command-Line Interfaces (CLI), or introducing human errors while manually breaking down policies into configuration commands. At the same time, its appeal also derives from the fact that it allows operators to express the same intent using different phrasings. However, the flexibility makes it challenging to generate configurations, which must capture operator intent in an unambiguous and accurate manner.

In this paper, we contribute to the ongoing IBN efforts by describing the design and implementation of LUMI, a new system that enables an operator “to talk to the network”, focusing on campus networks as a use case. That is, LUMI takes as input an operator’s intent expressed in natural language, correctly translates these natural language utterances into configuration commands, and deploys the latter in the network to carry out the operator’s intent. We designed LUMI in a modular fashion, with the different modules performing, in order: information extraction, intent assembly, intent confirmation, and intent compilation and deployment. Our modular design allows for easy plug-and-play, where existing modules can be replaced with alternative solutions, or new modules can be included. As a result, LUMI’s architecture is extensible and evolvable and can easily accommodate further improvements or enhancements.

In addressing the various challenges above, we make the following contributions:

Information extraction and confirmation. We build on existing machine learning algorithms for *Named Entity Recognition (NER)* [30] to extract and label entities from the operator’s natural language utterances. In particular, we implement NER using a chatbot-like interface with multi-platform support (§3) and augment the existing algorithm so that LUMI can learn from operator-provided feedback (§5).

Intent assembly and compilation. We introduce the *Network Intent Language (Nile)*, use it as an abstraction layer between natural language intents and network configuration commands for LUMI, and illustrate its ability to account for features critical to network management such as rate-limiting or usage quotas (§4). We also show how LUMI enables compilations of *Nile* intents to existing network programming languages (§6), such as *Merlin* [57].

Evaluation. We evaluate (§8) LUMI’s accuracy in information extraction, investigate LUMI’s ability to learn from operator-provided feedback and measure both the compilation and deployment times in a standard campus topology [5]. Using our own datasets consisting of synthesized intents as well as real-world intents derived from network policies published by 50 different campus networks in the US, we show that LUMI can extract entities with high precision, learn from the feedback provided by the operator, and compile and deploy intents in less than a second.

User study. In addition to an in-depth evaluation of LUMI, we also report our main findings of a small-scale user study, with 26 subjects (§9). The study was performed to get feedback from subjects on the perceived value of using natural language for network management with LUMI and soliciting user feedback during the intent confirmation stage.

Prototype. We implemented our prototype of LUMI using a combination of tools and libraries (*e.g.*, Google Dialogflow [28], *Scikit-learn* library [49]). The full implementation as well as all datasets used in our evaluation are available on the project’s website [39].

Together, the results of our evaluation and user study show that LUMI is a promising step towards realizing the vision of IBN of achieving fast, automated, and reliable policy deployment. By allowing operators to express intents in natural language, LUMI makes it possible for operators to simply talk to their network and tell it what to do, thus simplifying the jobs of network operators (*i.e.*, deploying policies) and also saving them time. While promising, developing LUMI into a full-fledged production-ready system poses new and interesting challenges on the interface of networking and NLP, which we detail in §10.

2 Lumi in a Nutshell

Figure 1 illustrates the high-level goal of LUMI with the intent example “*Hey, Lumi! Inspect traffic for the dorm*” and shows the breakdown of the workflow by which LUMI accomplishes

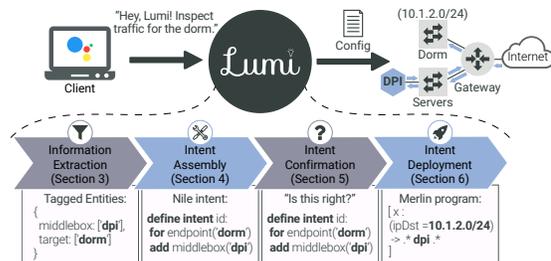


Figure 1: The four modules of LUMI.

the stated objective. Below, we provide a brief overview of the four key components that define this workflow (*i.e.*, the LUMI pipeline) and refer to the subsequent sections for a more in-depth description and design choices of each of these modules.

First, for the Information Extraction module (described in Section 3), we rely on machine learning to extract and label entities from the operator utterances and implement them using a chatbot-like conversational interface. The extracted entities form the input of the Intent Assembly module (described in Section 4), where they are used to compose a *Nile* network intent. *Nile* closely resembles natural language, acts as an abstraction layer, and reduces the need for operators to learn new policy languages for different types of networks. Then, as part of the Intent Confirmation module (described in Section 5), the output of the Intent Assembly module (*i.e.*, a syntactically-correct *Nile* intent) is presented to the network operator, and their feedback is solicited. If the feedback is negative, the system and the operator iterate until confirmation, with the system continuously learning from the operator’s feedback to improve the accuracy of information labeling over time. Finally, once the system receives confirmation from the operator, the confirmed *Nile* intent is forwarded to the Intent Deployment module. Described in Section 6, this module’s main task is to compile *Nile* intents into network configuration commands expressed in *Merlin* and deploy them in the network. In Section 6, we also explain why we picked *Merlin* as a target language over other alternatives.

3 Information Extraction

The main building blocks for LUMI’s Information Extraction module are a chatbot interface as the entry point into our system and the use of Named Entity Recognition (NER) [30] to extract and label entities from the operators’ natural language intents. Given the popularity of personal assistants, such as Google Assistant, Amazon’s Alexa or Apple’s Siri, our goal in providing a natural language interface for LUMI goes beyond facilitating the lives of traditional network operators and seeks to also empower users with little-to-no knowledge of how to control their networks (*e.g.*, home users).

Even in the case of the traditional user base of network operators, providing a natural language interface to interact with the system benefits teams composed of operators with

different levels of expertise or experience. This type of interface is particularly relevant in campus or enterprise networks with small groups and in developing countries where network teams are often understaffed and lack technical expertise. In short, while deploying a policy as simple as redirecting specific traffic for inspection can be a daunting task for an inexperienced operator, nothing is intimidating about expressing that same policy in natural language and letting the system worry about its deployment, possibly across multiple devices in the network.

Solving the NER problem typically involves applying machine learning (for extracting named entities in unstructured text) in conjunction with using a probabilistic graphical model (for labeling the identified entities with their types). Even though, in theory, NER is largely believed to be a solved problem [43], in practice, to ensure that NER achieves its purpose with acceptable accuracy, some challenges remain, including careful “entity engineering” (*i.e.*, selecting entities appropriate for the problem at hand) and a general lack of tagged or labeled training data.

Below, we first discuss the entity engineering problem and propose a practical solution in the form of a hierarchically-structured set of entities. Next, we describe in more detail the different steps that the NER process performs to extract named entities from a natural language intent such as “*Inspect traffic for the dorm*” and label them with their types. Finally, to deal with the problem caused by a lack of labeled training data, we describe our approach that improves upon commonly-used NER implementations by incorporating user feedback to enable LUMI to learn over time.

Table 1: Hierarchical set of entities defined in LumI.

Type	Entity Class
Common	@middlebox, @location, @group, @traffic, @protocol, @service, @qos_constraint, @qos_metric, @qos_unit, @qos_value, @date, @datetime, @hour
Composite	@origin, @destination, @target @start, @end
Immutable	@operation, @entity

3.1 Entity Selection

To ensure that NER performs well in practice, a critical aspect of specifying the underlying model is entity engineering; that is, defining which and how many entities to label in domain application-specific but otherwise unstructured text. On the one hand, since our goal with LUMI is to allow operators to change network configurations and manipulate the network’s traffic, the set of selected entities will affect which operations are supported by LUMI. As discussed in more detail in Section 4, LUMI-supported actions are dictated almost entirely by what intent constructions *Nile* supports. At the same time,

we would like to consider a generic enough set of entities to enable users to express their intents freely, independent of *Nile*. Moreover, the selected set of entities should also allow for easy expansion (*e.g.*, through user feedback; see Section 3.4 below) while ensuring that newly added entities neither introduce ambiguities nor result in unforeseen entries that might break the training model.

On the other hand, the selected set of entities directly influences the trained model’s accuracy, especially if the entities have been chosen poorly. Therefore, it is crucial to pick a set of entities that is at the same time rich enough to solve the task at hand and concise enough to avoid ambiguities that might hamper the learning process. For instance, one common source of uncertainty in network intents is highlighted with the two examples “*Block traffic from YouTube*” and “*Block traffic from the dorms*”. Here, the word ‘from’ appears in both intents but is used for two different purposes. While in the first example, it specifies a service, in the second example, it defines a location. If we choose to tag both entities (*i.e.*, “YouTube” and “dorms”) with the same entity type (*e.g.*, “location”) to avoid ambiguities, we lose information on what is being tagged (*i.e.*, service vs. location). However, if we simply use different entities for both cases (*e.g.*, “service” and “location”), we generate an ambiguity (*e.g.*, very similar phrasings produce entirely different results) that causes the accuracy of the NER model to decrease as similar example intents are encountered (*e.g.*, “*Block traffic from Twitter*”).

With these design requirements in mind, we defined the set of LUMI entities hierarchically and organized them into three different categories: common, composite, and immutable entities (see Table 1). Here, common entities form the bottom of the hierarchy, comprise raw textual values, and largely determine what LUMI can understand. For instance, the textual values in the common entity class @middlebox are network functions such as firewalls, packet inspection, and traffic shaping. The hierarchy’s intermediate level consists of composite entities. The entities in this class do not have any inherent nouns, verbs, or even synonyms associated with them; they only establish a relationship between common entities through the aggregation of prepositions. For instance, the composite entity class @origin consists of composite values such as “from @location” and “from @service”. Composite entities help avoid the ambiguity problem mentioned earlier. Finally, immutable entities make up the top of the hierarchy and form the core of LUMI. In particular, while the entity class @operation expresses the operations that *Nile* supports, the entity class @entity consists of a list of LUMI-supported common entities.

3.2 Entity Encoding: Bi-LSTMs

Figure 2 shows the overall NER architecture that we use in LUMI and illustrates the critical intuition behind NER; that is, finding named entities in unstructured text (*e.g.*, “*Inspect traffic for the dorm*” in Step 1 in Figure 2) and labeling them

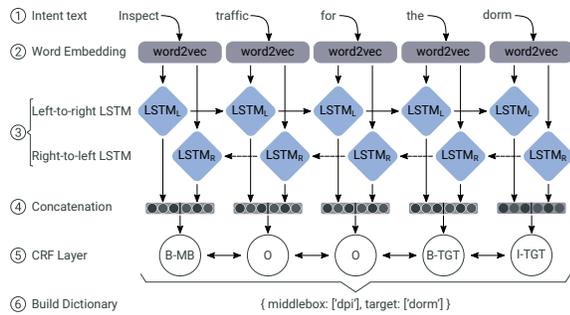


Figure 2: The NER architecture with Bi-LSTM (see Section 3.2) and CRF (see Section 3.3). The entity tags are abbreviated as MB for middlebox and TGT for target.

with their types (e.g., Step 6 in Figure 2). With respect to the machine learning part of the NER problem, standard approaches leverage Recurrent Neural Networks (RNN) [34]; i.e., a family of neural networks that processes arrays of data sequentially, where the output of one element is carried over as input to the next element in the sequence. However, RNNs typically require numerical inputs and not words. Therefore, when applied in the context of our text processing problem, each word of an intent that the operator expresses in natural language has to be first encoded into a numerical vector before an RNN can process it. Rather than using one-hot encoding [30], a simple encoding scheme that represents each encoded word as a vector of 0's and 1's, we rely in Step 2 (see Figure 2) on a more powerful approach that uses one-hot encoded vectors of words as input to a word embedding algorithm known as *word2vec* [44]. The *word2vec* algorithm uses a pre-trained mini-neural network that learns the vocabulary of a given language (English, in our case) and outputs a vector of decimal numbers for each word. As a neural network, *word2vec* can provide similar vector representations for synonyms, so that words with similar meanings (e.g., 'dormitories' and 'housing') have similar embedded vectors, which in turn allows an RNN to process them similarly.

Before processing the output of *word2vec*, we need to specify the type of RNN model in our architecture. The Long Short-Term Memory (LSTM) model has been a popular choice for text processing [25] as it can capture and carry over dependencies between words in a phrase or intent (e.g., to identify multi-word expressions). It also creates a context-full representation of each processed word as an encoded array of numerical values. However, to further enhance each word's context, in our LUMI design of the information extraction stage, we rely in Step 3 on an enhanced version of LSTM, the so-called Bi-LSTM model [16, 36]. The Bi-LSTM approach yields the best results for the English language in a majority of evaluated cases [63]. In a Bi-LSTM, a phrase is evaluated by two LSTMs simultaneously, from left-to-right and from right-to-left, and the outputs of the two LSTMs are then concatenated to produce a single output layer at a given position, as shown in Step 4.

3.3 Entity Labeling: CRFs

The labeling part of the NER problem consists of using the context-full encoded vectors of words to represent the "observed" variables for a type of probabilistic graphical models known as Conditional Random Fields (CRFs) [33], as shown in Step 5. CRFs are a widely-used technique for labeling data, especially sequential data arising in natural language processing. Their aim is to determine the conditional distribution $P(Y|X)$, where $X = \{x_1, x_2, \dots, x_n\}$ represents a sequence of observations (i.e., encoded vectors of words in a sentence) and $Y = \{y_1, y_2, \dots, y_m\}$ represents the "hidden" or unknown variable (i.e., NER tags or labels) that needs to be inferred given the observations. CRFs admit efficient algorithms for learning the conditional distributions from some corpus of training data (i.e., model training), computing the probability of a given label sequence Y given observations X (i.e., decoding), and determining the most likely label sequence Y given X (i.e., inference).

The technical aspects of the specific CRF model we use in this work are described in detail in LUMI's website [39] and show the flexibility afforded by CRF models to account for domain-specific aspects of labeling sequential data (e.g., accounting for the likelihood of one entity label being succeeded by another). However, irrespective of the specific CRF model used, the method outputs as the most likely tag for a given word the one with the highest probability among all tags. Specifically, at the end of Step 5, the result of the NER algorithm is provided in the form of named entities with IOB tagging [30]. In IOB tagging, each named entity tag is marked with an indicator that specifies if the word is the beginning (B) of an entity type or inside (I) of an entity type. If a word does not match any entity, then the algorithm outputs an (O) indicator (for "outside"). Note that by using IOB tagging, it is possible to distinguish if two side-by-side words represent a single entity or two completely different entities. For instance, without IOB-tagging, it would be impossible to tag an operation like "rate limiting" as one single named entity. Finally, we parse the IOB-tagged words resulting from the NER model, build a dictionary with the identified entities, and output it in Step 6 as the final result of LUMI's Information Extraction module.

3.4 NER and Learning

The described NER process is an instance of a supervised learning algorithm. It uses a corpus of training data in the form of input-output examples where input is an intent (i.e., phrase with named entities defined in LUMI and other words or non-entities), and an output is the list of named entities with IOB tagging (i.e., correct entity tags or labels). The primary training step consists of both adapting the weights of the Bi-LSTM model to extract the desired X vector and re-calculating the conditional distribution $P(X|Y)$ to infer the correct NER tags Y and may have to be repeated until convergence (i.e., Steps 3-5).

Note that retraining can be done each time the existing corpus of training data is augmented with new key-value pairs or with existing entities used in a novel context (*i.e.*, requiring a new tag). A basic mechanism for obtaining such new key-value pairs or for benefiting from the use of existing entities in a novel context is to engage users of LUMI and entice their feedback in real-time, especially if this feedback is negative and points to missing or incorrect pieces of information in the existing training data. By enticing and incorporating such user-provided feedback as part of the Intent Confirmation stage (see Section 5 for more details), LUMI’s design leverages readily available user expertise as a critical resource for constantly augmenting and updating its existing training data set with new and correctly-labeled training examples that are otherwise difficult to generate or obtain. After each new set of key-value pairs is obtained through user feedback, LUMI immediately augments the training corpus and retrains the NER model from scratch.

In light of the hierarchical structure of our LUMI-specific entities set, with user-provided feedback, we aim to discover and learn any newly-encountered common entities. Moreover, as the data set of common entities is augmented with new training instances, the accuracy of identifying composite entities also improves. With respect to the immutable entities, since the entity class *@operation* dictates what operations LUMI supports and understands, these operations cannot be learned from user-provided feedback. However, given the limited set of LUMI-supported operations, a relatively small training dataset should suffice to cover most natural language utterances that express these operations. As for the entity class *@entity*, being composed of a list of LUMI-supported common entities, it ensures that user-provided feedback can be correctly labeled.

4 Intent Assembly

Using a chatbot interface with NER capabilities as the front end of LUMI solves only part of the network intent refinement and deployment problem. For example, if a network operator asks a chatbot “Please add a firewall for the backend.”, the extraction result could be the following entities: *{middleboxes: ‘firewall’}*, *{target: ‘backend’}*. Clearly, these two key-value pairs do not translate immediately to network configuration commands. Assembling the extracted entities into a structured and well-defined intent that can be interpreted and checked for correctness by the operator before being deployed in the network calls for introducing an abstraction layer between natural language intents and network configuration demands.

To achieve its intended purpose as part of LUMI, this abstraction layer has to satisfy three key requirements. First, the operations supported by the abstraction layer’s grammar have to specify what entities to extract from natural language intents. Instead of trying to parse and piece together every possible network configuration from user utterances, we require that a predefined set of operations supported by the abstrac-

tion layer’s grammar guide the information extraction process. As a result, when processing the input text corresponding to a network intent expressed by the operator in natural language, we only have to look for entities that have a matching operation in the abstraction layer’s grammar, as those are the only ones that the system can translate into network configurations and subsequently act upon.

A second requirement for this abstraction layer is to allow for easy confirmation of the extracted intents by the operators by having a high level of legibility. We note that requiring confirmation does not burden the operators in the same manner that requiring them to express their network intents directly in the abstraction layer’s language would. For instance, it is well-known that a person who can understand a written sentence cannot necessarily create it from scratch (*i.e.*, lacking knowledge of the necessary grammar).

Finally, the abstraction layer is also required to allow different network back-ends as targets given that a wide range of candidate network programming languages [6, 11, 21, 31, 46, 54, 57] exist, and none of them seem to have been favored more by operators or industry yet. Using an abstraction layer on top of any existing languages allows us to decouple LUMI from the underlying technology, so we can easily change it if a better option arises in the future.

4.1 Nile: Network Intent Language

To satisfy the above requirements, in our design of the Intent Assembly module (*i.e.*, stage two of the LUMI pipeline), we rely on the Network Intent Language (*Nile*) to serve as our abstraction layer language. In a previous work [29], we proposed an initial version of *Nile* that provided minimal operation support for intent definition and focused primarily on service-chaining capabilities. Here, we extend this original version to cover crucial features for network management in real-world environments (e.g., usage quotas and rate-limiting). Closely resembling natural language, the extended version of *Nile* has a high level of legibility, reduces the need for operators to learn new policy languages for different types of networks, and supports different configuration commands in heterogeneous network environments. Operationally, we designed this module to ingest as input the output of the information extraction module (*i.e.*, entities extracted from the operator’s utterances), assemble this unstructured extracted information into syntactically-correct *Nile* intents, and then output them.

Table 2: Overview of *Nile*-supported operations.

Operation	Function	Required	Policy Type
from/to	endpoint	Yes	All
for	group/endpoint/service/traffic	Yes	All
allow/block	traffic/service/protocol	No	ACL
set/unset	quota/bandwidth	No	QoS
add/remove	middlebox	No	Service Chaining
start/end	hour/date/datetime	No	Temporal

Table 2 shows the main operations supported by our extended version of *Nile*, and the full grammar of *Nile* is made available in LUMI’s website [39]. Some of the operations have opposites (e.g., allow/block) to undo previously deployed intents and enable capturing incremental behaviors stated by the operators. Some operations in a *Nile* intent are mandatory, such as **from/to** or **for**. More specifically, an operator cannot write an intent in *Nile* without stating a clear target (using **for**) or an origin and a destination (using **from/to**) when the direction of the specified traffic flow matters.

To enforce the correct syntax of the assembled intent, we leverage the *Nile* grammar to guarantee that the intent contains the required information for correct deployment. If the grammar enforcement fails due to the lack of information, the system prompts the operator via the chatbot interface to provide the missing information. Assume, for example, that the operator’s original intent stated “Please add a firewall.”, without providing the target of the intent. Since specifying a target is required according to the *Nile* grammar, the module will not attempt to construct a *Nile* program but will instead interact with the operator to obtain the missing information.

4.2 *Nile* Intents: An Example

With *Nile*, we can express complex intents intuitively. For example, an input like “Add firewall and intrusion detection from the gateway to the backend for client B with at least 100mbps of bandwidth, and allow HTTPS only” is translated to the *Nile* intent shown in Listing 1. The **group** function is used as a high-level abstraction for clients, groups of IP addresses, VLANs, or any other aggregating capacity in low-level network configurations. Note that the IDs provided by the operator must be resolved during the compilation process, as they represent information specific to each network. This feature of the language enhances its flexibility for designing intents and serving as an abstraction layer. The example illustrates how *Nile* provides a high-level abstraction for structured intents and suggests that the grammar for *Nile* is expressive enough to represent many real-world network intents.

```
define intent qosIntent:
  from endpoint('gateway')
  to endpoint('database')
  for group('B')
  add middlebox('firewall'), middlebox('ids')
  set bandwidth('min', '100', 'mbps')
  allow traffic('https')
```

Listing 1: *Nile* intent example.

5 Intent Confirmation (Feedback)

The major challenge with using natural language to operate networks (e.g., as part of IBN) is that the method is inherently ambiguous. There are many different forms in which an operator can express the same intent in different network environments in natural language. Despite recent advances, natural language processing is prone to producing false positives or false negatives, resulting in incorrect entity tagging,

leading to deploying incorrect network configuration commands. However, to be of practical use, network configurations are required to be unquestionably unambiguous and correct-by-construction.

One approach to address this challenge is to create an extensive dataset with training phrases and entities. However, it is unrealistic to expect that such a dataset will cover every possible English language (or any other language for that matter) example of phrase construction with domain-specific entities. Operators are free to use terms or expressions that the system has never encountered in the past. However, without proper safeguards, any such new phrase will likely result in misconfigurations of the network. An alternative approach to implementing a reliable intent deployment process is through “learning from the operator.” Here, the basic idea is to leverage the operators’ existing knowledge by requesting their feedback on the information extracted from natural language intents. Then, in the case of negative feedback, it is critical to engage with the operators to identify missing or incorrect pieces, include this such acquired new knowledge as additional phrases or entities in the original training dataset, and retrain the original learning model.

Our solution to deal with the ambiguity inherent in using natural language to express network intents follows the learning approach and leverages the chatbot interface implemented as part of our first module. In particular, in designing the intent confirmation module for realizing stage three of the four-stage LUMI pipeline, we require the output of the intent assembly module (i.e., syntactically-correct *Nile* intents) to be confirmed by the operator. When presented with assembled intents that result in false positives or negatives, the operator is asked to provide feedback that we subsequently use to augment the original training dataset with new labeled data; that is, LUMI is capable of learning new constructs over time, gradually reducing the chances of making mistakes. While this interaction may slow down initial intent deployments until LUMI adapts to the operator’s usage domain, it is essential to guarantee reliable intent refinement and deployment.

Extracting pertinent information from user feedback also requires identifying specific entities in the user text, similarly to extracting entities from an input network intent. To this end, LUMI uses the same NER model for both tasks, relying primarily on the immutable *@entity* for extracting which entity class is being augmented and what value is being added. Relying on the same NER model also requires us to train the model to identify and extract entities in the received user feedback. However, since we limit LUMI’s interactions with the user to answering simple questions, processing user feedback does not require a large set of training samples.

To reduce an operator’s need for technical knowledge during this intent confirmation stage, we opted for supporting feedback interactions that induce the operator to provide the information that LUMI needs (i.e., offering suggestions and asking complementary questions). Also, to provide operators

with more flexibility and not insist that they have to use specific keywords, this module compares the entities and values provided by the operators with synonyms already in the training dataset. Figure 3 illustrates a case where, due to a lack of training, LUMI misses an entity in the input text, and the confirmation mechanism lets operators easily catch this mistake and provide the necessary feedback for LUMI to learn. Also, note that our design of this module is conservative in the sense that operator feedback is requested for each assembled intent, irrespective of the level of confidence that LUMI has concerning its accuracy.

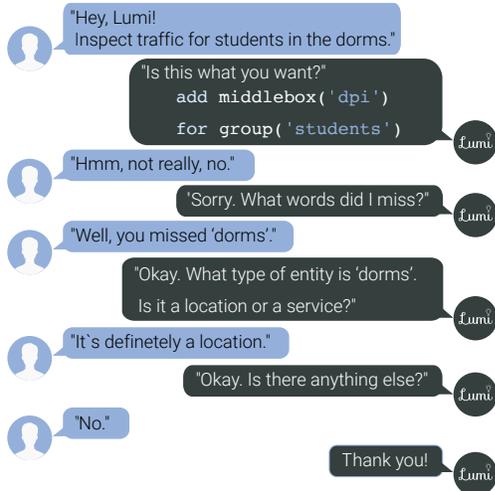


Figure 3: LUMI’s feedback mechanism in action.

6 Intent Deployment

The fourth and last stage of LUMI compiles the operator-confirmed *Nile* intents into code that can be deployed on the appropriate network devices and executes the original network intent expressed by the operator in natural language. Fortunately, the abstraction layer provided by *Nile* enables compilations to a number of different existing network configurations, including other policy abstractions languages such as *Merlin* [57], *OpenConfig* [48], *Janus* [2], *PGA* [51], and *Kinetic* [31].

For our design of LUMI’s Intent Deployment module, we chose to compile structured *Nile* intents into *Merlin* programs. We picked *Merlin* over other alternative frameworks because of its good fit with *Nile*, the network features it supports, its performance, and the availability of source code. Also, given that none of the existing network programming languages natively supports all features proposed in *Nile*, we opted for the one that had the largest intersection of supported operations. In the process, we mapped each *Nile* feature to a corresponding *Merlin* feature.

Resolving logical handles. Logical handles in *Nile* intents are decoupled from low-level IP addresses, VLAN IDs and IP prefixes, which LUMI now resolves (e.g., `dorm` → `10.1.2.0/24`) using information provided during the bootstrap process. ACLs rules are resolved similarly. Once LUMI

produces *Merlin* programs with resolved identifiers (i.e., VLAN IDs, IPs and prefixes), compilation to corresponding OpenFlow rules is handled by *Merlin*.

Temporal constraints and QoS. As *Merlin* does not support temporal policies, LUMI stores every confirmed *Nile* intent so that it can install or remove device configurations according to times and dates defined by the operator. We achieve quota restrictions (not natively supported by *Merlin*) by relaying all traffic marked with a quota requirement to a traffic-shaping middlebox, taking advantage of *Merlin*’s support for middlebox chaining. Other QoS policies, such as rate-limiting, are already supported in *Merlin*.

Middlebox chaining. LUMI focuses on middlebox chaining, i.e., correctly relaying traffic specified in the intents through a specified middlebox. Since the actual configuration of each middlebox is currently done outside of LUMI, LUMI can handle chaining policies associated with *any* middlebox type, virtual or physical, compilation for which is straightforward since *Merlin* natively supports middlebox chaining.

7 Implementation

We implemented LUMI’s prototype using different tools and libraries for each module of the LUMI-pipeline. For the chatbot interface and NER, we used Google Dialogflow [28] because of its conversational interface and multi-platform integration. However, these stages can be easily implemented with other open-source NLP frameworks (e.g., SpaCy [26] and Stanford CoreNLP [41]) or machine learning toolkits such as Scikit-learn [49] and Keras [18]. We exported the Dialogflow implementation of LUMI as JSON files and uploaded them to GitHub [39], so other researchers can build on our work and reproduce our results.

We implemented the WebHook service that Dialogflow calls with the extracted entities in Python. This service builds the *Nile* intents and interacts with the chatbot interface if necessary. We implemented the compilation of *Nile* programs into *Merlin* programs and the deployment of the resulting *Merlin* program as a separate Python RestAPI service that the previous WebHook service calls after the operator confirms that the intent is correct. We developed this module as a separate service so that it can be deployed on a local server with access to the network controller. All the other modules can be deployed on a cloud server. The full implementation of LUMI, comprising over 5,451 lines of Python code and 1,079 lines of JavaScript and HTML, a working demo, and all datasets used in the course of this work are publicly available [39].

8 Evaluation

In this section, we first evaluate the accuracy of our Information Extraction module to show that LUMI extracts entities with high precision and learns from operator-provided feedback. We then show that LUMI can quickly compile *Nile* intents into *Merlin* programs and deploy them. To assess the accuracy of the Information Extraction module, we use the standard metrics Precision, Recall, and F1-score. For the

evaluation of the Intent Deployment stage, we measure the compilation time for translating *Nile* intents into Merlin statements and their deployment time.

8.1 Information Extraction

Evaluating systems like LUMI is challenging because of (i) a general lack of publicly available datasets that are suitable for this problem space (several operators we contacted in industry and academia gave proprietary reasons for not sharing such data), and (ii) difficulties in generating synthetic datasets that reflect the inherent ambiguities of real-world Natural Language Intents (NLIs).

To deal with this problem, we created two hand-annotated datasets for information extraction. The dataset *alpha* is “semi-realistic” in the sense that it is hand-crafted, consisting of 150 examples of network intents that we generated by emulating an actual operator giving commands to LUMI. In contrast, the *campi* dataset consists of real-world intents we obtained by crawling the websites of 50 US universities, manually parsing the publicly available documents that contained policies for operating their campus networks, and finally extracting one-phrase intents from the encountered public policies. From those 50 universities, we were able to extract a total of 50 different network intents. While some universities did not yield any intents, most universities published network policies related to usage quotas, rate-limiting, and ACL, and we were able to express all of them as *Nile* intents. We manually tagged the entities in each of these 200 intents to train and validate our information extraction model.

We used both datasets, separately and combined, to evaluate our NER model, with a 75%-25% training-testing random split. The small size of each dataset precludes us from performing conventional cross-validation. Table 3 shows the results for the *alpha* dataset, for the *campi* dataset, and for a combination of the two and illustrates the high accuracy of LUMI’s information extraction module. Given the way we created the training examples for the *alpha* dataset, the excellent performance in terms of Precision, Recall, and F1-score is reassuring but not surprising. In creating the intent examples, we paid special attention to extracting all the entities defined in LUMI (see Section 3.1) and also creating multiple intents for each entity class.

Table 3: Information extraction evaluation using the *alpha* and *campi* dataset.

Dataset	# of Entries	Precision	Recall	F1
<i>alpha</i>	150	0.996	0.987	0.991
<i>campi</i>	50	1	0.979	0.989
<i>alpha + campi</i>	200	0.992	0.969	0.980

At the same time, despite the largely unstructured nature and smaller number of intent examples in the *campi* dataset, the results for that dataset confirm the above observation.

Even though the example intents in this case were not designed with the NER model in mind, LUMI’s performance remains excellent and is essentially insensitive to the differences in how the intent examples were generated. We attribute this success of LUMI at the information extraction stage to both continued advances in using machine learning for natural language processing and the fact that the complete set of LUMI-defined entities is relatively small and at the same time sufficiently expressive.

8.2 Intent Confirmation and Feedback

To evaluate the impact of operator-provided feedback on LUMI’s ability to learn, we first trained our NER model using 75% of the combined *alpha* and *campi* datasets (*i.e.*, a total of 150 training examples) and then used the remaining 25% of examples (*i.e.*, a total of 50 test entries) as new intents that we presented LUMI in random order. We fed each of the 50 test intents into the NER model for information extraction and evaluated the Precision and Recall on a case-by-case basis. If a new intent generated False Positives or False Negatives, we inserted the intent into NER’s existing training dataset, alongside the pre-tagged entities, mimicking the operator’s feedback given during the Intent Confirmation stage.

The results for this experiment (Precision, Recall and F1-score) are shown in Figures 4a and 4b. Since each point in the plots represents the Precision/Recall for one specific test sample rather than for the global model, the depicted values fluctuate as test samples are checked one after another. As can be seen, while Precision quickly reaches and then stays at 1.0, the Recall metric dips each time the model encounters an entity or construct it has not yet seen (*i.e.*, resulting in a False Negative). However, as more of these examples become part of NER’s training data through the feedback mechanism (and because of re-training of the model after each new example is added), these dips become less frequent. Out of the 50 test intents, only eight resulted in a dip in Recall; that is, were used as feedback. Note, however, that the cases where the model does not identify an entity are precisely the situations where feedback is most informative and enables the model to learn for the benefit of the users.

To assess how often a user has to rely on the feedback mechanism, we repeated our experiment 30 times, each time with a different 75-25 training-testing split. The resulting mean values for Precision and Recall are shown in Figures 4c and 4d, with corresponding 95% confidence intervals. As expected, over the 30 repetitions, both Precision and Recall remain close to 0.99, with very small fluctuations. And just as in the previous experiment, whenever there is a significant variation in Precision or Recall (*i.e.*, large confidence intervals), we added that particular intent example to NER’s latest training dataset and retrained the model. We attribute the fact that about 20% of the test examples were used as feedback to the small size of our training dataset, but argue that having only this many feedbacks is a positive outcome.

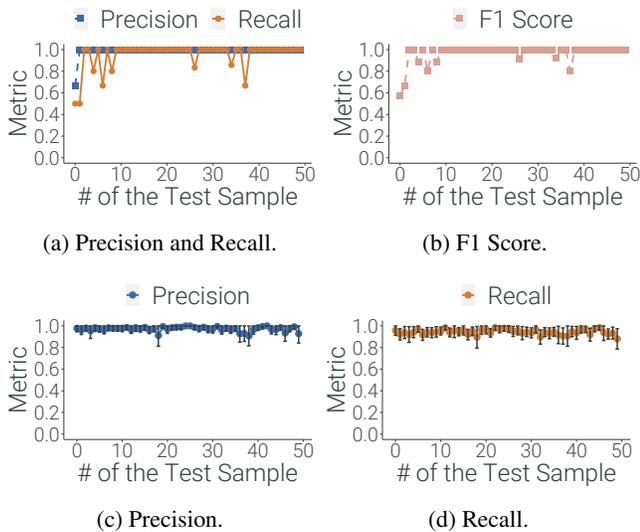


Figure 4: Feedback impact on information extraction.

8.3 Intent Deployment

To evaluate the deployment capabilities of LUMI, we compile and deploy five categories of *Nile* intents, with an increasing level of complexity: middlebox chaining, ACL, QoS, temporal, and intents with mixed operations. The last category of intents mixes *Nile* operations with distinct goals, which we use to evaluate the deployment of more complex intents. We generated a dataset with 30 intents per category, totaling 150 different intents, and measured the mean compilation and deployment time of each category. We ran this experiment on a generic campus network, with approximately 180 network elements. We relied on the Mininet [35] emulator to perform the experiments. The results are given in Table 4. While deployment time necessarily depends on the network environment, in our setting, we consistently measured sub-second combined compilation and deployment times.

Table 4: Compilation and deployment time for five categories of *Nile* intents.

Intent Type	Compilation Time (ms)	Deployment Time (ms)
Middlebox chaining	4.402	110
ACL	3.115	112
QoS	3.113	136
Temporal	4.504	111
Mixed	4.621	1030

9 User Study

To evaluate LUMI’s ability to work in a real-world environment rather than with curated datasets of intents, we designed and carried out a small-scale user study. Specifically, we wanted to assess three critical aspects of our system: (i) How well does the information extraction process work with actual humans describing their intents in different forms and phrasings? (ii) How often is it necessary for operators to provide feedback for LUMI while using the system? and (iii) Com-

pared to existing alternative methods, what is the perceived value of a system like LUMI that leverages natural language for real-time network management?

In this section, we describe the experiments we conducted, the participants’ profiles, and the obtained results. We set up the user study as an online experiment that users could access and participate in anonymously. To select participants for the user study from different technical backgrounds and still keep their anonymity, we distributed a link to the online user study in mailing lists of both networking research groups and campus network operators. According to the guidelines of our affiliated institution, due to the fully anonymous nature of the experiment, no IRB approval was required to conduct this study, so this work does not raise any ethical issues.

9.1 Methodology

Participating users were asked to fill out a pre-questionnaire (e.g., level of expertise, degree of familiarity with policy deployment, and use of chatbot-like interfaces) and then take on the role of a campus network operator by performing five specific network management tasks using LUMI. Based on the information these users provided in their pre-questionnaire, we had participants from three different continents: the Americas (88.5%), Europe (7.7%), and Asia (3.8%).

Each of the tasks required the user to enforce a specific type of network policy: (i) reroute traffic from the internet to an inspection middlebox; (ii) limit the bandwidth of guest users using torrent applications; (iii) establish a data usage quota for students in the university dorms; (iv) block a specific website for students in the labs, and (v) add a daily temporal bandwidth throttle to the server racks from 4pm to 7pm. Every interaction users had with LUMI was logged to a database for post-analysis.

After finishing the tasks, users were asked to complete a post-questionnaire (e.g., number of completed tasks, the perceived value of LUMI, and comments on its usability). The complete set of management tasks presented to the users and all the results are available on the LUMI’s website. Out of the 30 participants, four did not complete the online questionnaires and were excluded from the study, leaving a total of 26 subjects. Figure 5 shows a breakdown of the user profiles by type of job, level of experience with network management, and familiarity with chatbot-like interfaces.



Figure 5: Subjects profiles.

9.2 Information Extraction and Feedback

To assess the accuracy of our Information Extraction module, we use the number of tasks each participant concluded. For

completing any given task, a specific set of labeled entities was required to build the correct *Nile* intent. Hence, each user’s number of completed tasks reflects how accurately LUMI identified the entities in the input texts. The results in the left part of Figure 6 show that most users completed either 5/5 or 4/5 tasks. Some examples of successful intents for each task can be found on LUMI’s website [39]. An analysis of the users’ interactions with the system revealed that LUMI had trouble understanding temporal behavior (e.g., “from 4pm to 7pm”), likely due to a lack of such training examples. This issue prevented some users from completing all five tasks. One user could not complete any task, reportedly because of an outage in the cloud provider infrastructure used to host LUMI.

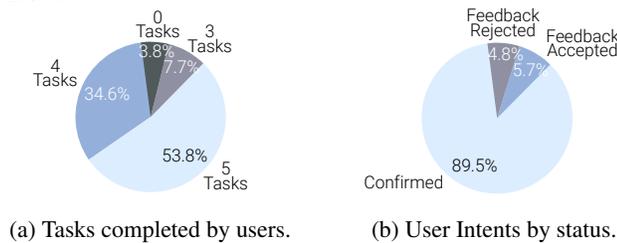


Figure 6: LUMI information extraction and feedback.

To evaluate the value of LUMI’s feedback mechanism as part of the Intent Confirmation module, we considered all intents that the 26 users generated and checked how many were confirmed and how many were rejected. If an intent was rejected, the user could provide feedback to correct it, thus improving LUMI. Such a corrected intent could then once again be accepted or rejected by the user. The right-hand side of Figure 6 gives the breakdown of the intents and shows that, most of the time, LUMI correctly interpreted the users’ intents; in the few times feedback was needed, LUMI was able to correct and learn more often than not. This result is encouraging given the somewhat limited amount of data with which LUMI was trained.

On further analysis of the interactions that users had with LUMI, we observed that the feedback mechanism worked as expected in cases where the participants used a different or unusual phrasing of their intents. For instance, one user expressed the intent for task 3 in an unstructured manner, as “*student quota dorms 10GB week*”, in which LUMI was not able to recognize the word “*dorms*” as a *@location* entity. However, the user then provided feedback that was successful in correcting the original *Nile* intent.

One concrete example where the feedback was unsuccessful happened in task 3, with a user that typed the intent “*Lumi, limit students to 10GB maximum download per week in dorms*”. LUMI was only trained to recognize phrases of the form “10GB per week”, and the additional text in between resulted in LUMI being unable to recognize the user’s phrase. When asked what information LUMI had missed, the user provided feedback indicating that “10gb/wk” was an entity class and “Gb per week” was the value, instead of labeling

“Gb per week” as a *@qos_unit* entity. We note that LUMI is an initial prototype, and such cases can be avoided in the future by improving the clarity of suggestions LUMI makes to the user, and by including sanity checks on user inputs.

9.3 Users Reactions and Usability

In the post-questionnaire, we asked the users to comment on LUMI’s usability and overall merit by answering three questions: (i) How easy was it to use LUMI to configure the network? (ii) Compared to traditional policy configuration, how much better or worse was using LUMI instead? and (iii) Would they rather use LUMI to enforce policies or conventional network configuration commands. Figures 7, 8 and 9 summarize the users’ responses, broken down by expertise in network management. The results show that the participants’ overall reaction to LUMI was very positive, with most of them stating that they would either use LUMI exclusively or, depending on the tasks, in conjunction with configuration commands. Note that the expert users who identified themselves as campus network operators all had a positive reaction to LUMI. Overall, among all different levels of expertise, 88.5% of participants stated they would rather use LUMI exclusively or in conjunction with configuration commands.

We also asked participants to provide insights they had that could help us improve the implementation and design of LUMI. One important feedback we received was the lack of support for querying the network state. For example, one participant stated:

“*Many network management tasks are about monitoring something or figuring out what’s happening, not just installing a new policy...*”

While LUMI was not designed with this goal in mind, we do not foresee any major NLP challenge to incorporate such features into it, as the entity set could be extended to cover this use case. Acquiring the state information from the network requires further investigation, but LUMI’s modular design makes it simpler to plug in a new module or an existing one [13] to query the network devices. This would enable LUMI to “understand” changes made through other tools, as LUMI will likely co-exist with different management applications in real deployments. Overall, the feedback received from participants was positive and highlighted the promise and value of LUMI.

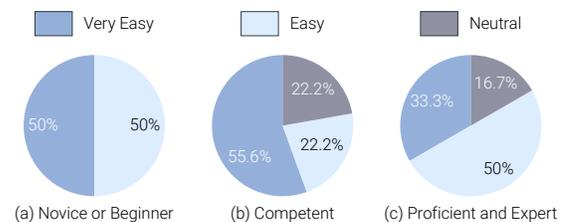


Figure 7: User reaction to LUMI’s usability, by expertise on network management.

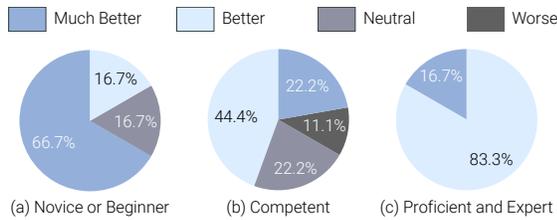


Figure 8: User reaction to LUMI's when compared to traditional network configuration, by expertise on network management.

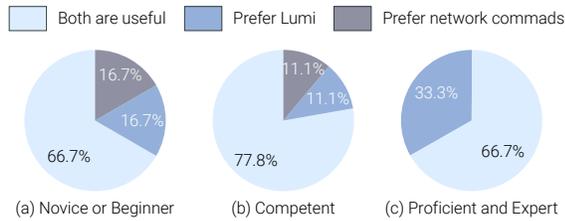


Figure 9: User reaction to LUMI's when compared to traditional network configuration, by expertise on networking.

10 Ongoing Work and Open Problems

While we consider LUMI to be a promising and necessary step towards fully realizing IBN, several challenges remain and are part of our ongoing work.

10.1 Ambiguities in natural language policies

NLP policies have the potential for ambiguities. In exploring this issue further, we extracted pairs of intents from our *campi* dataset and generated 213 pairs in all. Furthermore, we adapted existing NLP efforts on contradiction detection in general text [24, 37, 40, 42, 53, 55, 60] by developing a Random Forest Classifier trained to classify pairs of network intents based on contradiction indicators (features) found between the two input intents. The classifier flagged 9 cases with potential contradictions.

Manual inspection of all 213 intent pairs indicated that most of these cases were benign. They typically corresponded to cases where universities expressed policies that depended on a user's total traffic usage over different time periods (e.g., a 10 GB weekly limit vs a 5 GB daily limit). In these cases, the two policies could be applied in any order with no negative consequences. One interesting case, however, was a campus that expressed two different policies at different locations on their website. The first policy indicated H323 video conferencing was allowed by the University firewall, while the second indicated MSN audio and video communications were not allowed. This is a case where the relative precedence between the two policies impacts their joint effect.

Part of our ongoing work is to combine LUMI with formal methods to help detect ambiguities that are potentially of concern. Specifically, translating NLP policies into LUMI intents enables the use of automated methods that can check

whether the impact of applying two policies is sensitive to their relative ordering, but also provides opportunities to use methods for detecting policy conflicts [2, 51].

10.2 Deploying Lumi in a production network

The initial design of LUMI was aimed at solving management problems that arise in a Campus network environment. We have been engaged in discussions with operators of our campus network regarding validating LUMI in production. Below, we discuss some of the issues raised by the operators and outline challenges and potential solutions.

Co-existing with current technologies. Most campus networks consist of legacy network equipment from multiple vendors with vendor-specific configuration interfaces. Getting LUMI "production-ready" requires developing a *Nile* compiler that can accommodate this diversity in legacy devices. Since the *Nile* abstraction offers isolation from the system's interface and minimizes the need for changes in the early stages of the LUMI pipeline, it is well-suited for Open-Config [48], a vendor-neutral model for network management that is supported by an increasing number of devices.

Extending LUMI for other use-cases. While we have focused on Campus networks, deploying LUMI in other environments may require extending its feature set. For example, unlike the Campus networks we have access to, multi-tenant data-center networks may use VXLANs or NVGRE as solutions to scalably share network infrastructure between tenants. However, LUMI is easily extended to support such features, and we illustrate the four generic steps required for such extension with the VXLAN example. In a first step, one must decide the abstraction level in which LUMI should handle natural language text. Consistent with LUMI's design philosophy, for VXLAN support, a high-level intent could be "*Block incoming traffic for tenant A*", where "*tenant A*" refers to a specific VXLAN Network Identifier (VNI). Next, LUMI's training dataset has to be augmented appropriately, the *Nile* language must be extended with new keywords (e.g., a `tenant('A')` operator for the VXLAN example), and the *Nile* compiler has to be instrumented to handle the new set of configurations. In the case of VXLAN, similarly to VLANs, tenants names must be mapped to VNIs. Lastly, since OpenFlow switches support VXLANs, all that is needed is to extend *Merlin* to allow matching traffic based on VNI for each tenant.

Creating sandbox environments. Migrating LUMI to production requires that operators have trust in LUMI. Based on our discussions with operators, a mirrored sandbox environment [3] could be a good starting point.

10.3 How to verify if Lumi is correct?

We discuss potential sources of errors in each stage of the LUMI pipeline and possible solution approaches.

Translating human language to *Nile* intents. Information extraction from natural language is inherently prone to errors. LUMI alleviates this problem by asking operators for

feedback and using their responses to check if the extracted information was correct. Over the longer term, we plan to leverage ongoing ML research efforts that focus on making ML models more robust and secure so they can be deployed in security- and safety-critical settings such as production network environments [9, 27].

Compiling *Nile* intents. Recent works on formal network verification [1, 10, 50] provide sub-second verification of waypointing, reachability, and isolation properties and are well suited for verifying configurations generated by LUMI-compiled intents. LUMI supports time-constrained intent deployment as well as QoS features. Despite recent work on verifying such properties (e.g., [31, 58]), more advances may be needed in the area, including the possible adaption of existing verification techniques to a LUMI-specific setting.

Post-deployment behavior monitoring. Ultimately, we envision the LUMI pipeline shown in Figure 1 to include a monitoring module for verifying that the deployed configurations respect the intents produced by the refinement process and achieve the objective(s) that the operator expressed (in natural language) in the first place. By monitoring both the traffic and configurations of specific devices affected by a deployed intent, such a module would allow operators to query at any time if the deployed intent produced the desired network behavior [22], thereby improving the operators' trust in relying on LUMI. However, deciding which traffic, devices, and properties to monitor will require instrumenting networks with an unprecedented level of control that is currently only possible by leveraging the latest programmable data plane technologies [8, 23]. At the same time, the development of such a module can be viewed as a first step towards realizing the vision of self-driving networks [12, 19, 45].

11 Related Work

Natural language in networking. Very few prior works use natural language to interact with the network. In [13], the authors present *Net2Text*, a system that allows network operators to query network-wide forwarding behaviors using natural language, but it does not allow operators to configure the network. Alsudais *et al.* [4] proposes using natural language to deploy network intents. It uses the Stanford CoreNLP Parts-of-Speech (POS) Tagger [61] to parse and structure the input text but does not cover NLP aspects relevant to LUMI such as intent confirmation for user feedback and learning over time.

Network programming languages. Recent works on IBN feature several intent languages, frameworks, and compilers to efficiently deploy intents in network devices and middleboxes [2, 6, 21, 31, 51, 54, 57, 59]. At the same time, Cocoon [54] introduces a framework to guarantee the correctness of SDN programs that resembles our approach, but it uses first-order logic. With LUMI, we examine the use of machine learning to convert natural language intents into lower-level configurations without the need for using a specific programming language.

Natural language processing. Information extraction has been addressed with different methods and techniques, including (i) only CRF models [20]; (ii) character-level embeddings instead of whole words [34], and (iii) rule-based approaches [15]. Yet, recent studies [63] show the benefits of the approach considered as part of LUMI. To our knowledge, the problem of using natural language for management tasks has not received much attention in the networking domain.

IBN. INSpIRE [56] focuses on intents related to security middleboxes and uses a refinement process to determine which middleboxes should compose a service chain to fulfill an intent. Cheminod *et al.* [14] propose an automatic process for refining, deploying, and enforcing ACL policies that use set notation for policy specification. PGA [51] applies a graph-based abstraction to compose high-level policies and deploy them in SDN networks. Janus [2] extends PGA to support policies with QoS requirements, mobility, and temporal dynamics. However, these proposals differ from LUMI as they are not concerned with using natural language or obtaining feedback from the network operator.

12 Conclusions

In this paper, we propose LUMI, a novel end-to-end intent refinement and deployment system that allows operators to express their intents in natural language and then check and confirm the intents before deploying them in the network. By demonstrating that LUMI can successfully deal with a wide range of network policies, this paper represents a promising step towards realizing the vision of intent-based networking with natural language. Still, much work remains. For example, while our design choices for LUMI's different modules resulted in a working prototype, other features might be necessary for a production-ready version of the system. However, LUMI's modular design can readily accommodate such improvements. Also, since LUMI in its current form is mainly intended for use in campus networks, supporting other environments (e.g., home or enterprise networks) will most likely require that the set of *Nile* operations and functions (and in turn the set of LUMI entities) be judiciously extended.

Acknowledgments

We thank our shepherd Costin Raiciu and the anonymous reviewers for their valuable feedback. We thank Jennifer Rexford, Hyojoon Kim, Shir Landau-Feibish, and Ross Teixeira for their feedback on earlier drafts of this paper. We also thank the anonymous participants of our user study for their contributions and insights. This work was supported in part by the Brazilian National Research and Educational Network (RNP), the Brazilian Federal Agency for Support and Evaluation of Graduate Education (CAPES), the Brazilian National Council for Scientific and Technological Development (CNPq) procs. 423275/2016-0, 312392/2017-6, 142089/2018-4, INCT InterSCity, and FAPESP procs. 2018/23085-5, 2020/05183-0, and 2015/24494-8, and the US National Science Foundation Grant FMITF 1837023.

References

- [1] A. Abhashkumar, A. Gember-Jacobson, and A. Akella. Tiramisu: Fast and General Network Verification. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, Santa Clara, CA, February 2020. USENIX Association.
- [2] A. Abhashkumar, J. Kang, S. Banerjee, A. Akella, Y. Zhang, and W. Wu. Supporting Diverse Dynamic Intent-based Policies Using Janus. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, pages 296–309, New York, NY, USA, 2017. ACM.
- [3] R. Alimi, Y. Wang, and Y. R. Yang. Shadow Configuration as a Network Management Primitive. In *Proceedings of the Annual Conference of the ACM SIGCOMM*, SIGCOMM '08, page 111–122, New York, NY, USA, 2008. ACM.
- [4] A. Alsudais and E. Keller. Hey network, can you understand me? In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 193–198, May 2017.
- [5] A. Amokrane, R. Langar, R. Boutaba, and G. Pujolle. Flow-Based Management For Energy Efficient Campus Networks. *IEEE Transactions on Network and Service Management*, 12(4):565–579, December 2015.
- [6] C. J. Anderson, N. Foster, A. Guha, J. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic Foundations for Networks. *SIGPLAN Not.*, 49(1):113–126, January 2014.
- [7] J. Apostolopoulos. Improving Networks with Artificial Intelligence, Oct 2020. <https://blogs.cisco.com/networking/improving-networks-with-ai>.
- [8] R. B. Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher. PINT: Probabilistic In-Band Network Telemetry. In *Proceedings of the Annual Conference of the ACM SIGCOMM*, SIGCOMM '20, page 662–680, New York, NY, USA, 2020. ACM.
- [9] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi. Measuring Neural Net Robustness with Constraints. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.
- [10] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A General Approach to Network Configuration Verification. In *Proceedings of the Annual Conference of the ACM SIGCOMM*, SIGCOMM '17, pages 155–168, New York, NY, USA, 2017. ACM.
- [11] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *Proceedings of the Annual Conference of the ACM SIGCOMM*, SIGCOMM '16, pages 328–341, New York, NY, USA, 2016. ACM.
- [12] M. Behringer, M. Pritikin, S. Bjarnason, A. Clemm, B. Carpenter, S. Jiang, and L. Ciavaglia. Autonomic Networking: Definitions and Design Goals. Rfc 7575, RFC Editor, June 2015.
- [13] R. Birkner, D. Drachler-Cohen, L. Vanbever, and M. Vechev. Net2Text: Query-Guided Summarization of Network Forwarding Behaviors. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pages 609–623, Renton, WA, 2018. USENIX Association.
- [14] M. Cheminod, L. Durante, L. Seno, F. Valenza, and A. Valenzano. A comprehensive approach to the automatic refinement and verification of access control policies. *Computers & Security*, 80:186–199, 2019.
- [15] L. Chiticariu, M. Danilevsky, Y. Li, F. Reiss, and H. Zhu. SystemT: Declarative Text Understanding for Enterprise. In *Proceedings of the 2018 Conference of the North American Chapter of the ACL: Human Language Technologies, Volume 3*, pages 76–83. ACL, 2018.
- [16] J. Chiu and E. Nichols. Named Entity Recognition with Bidirectional LSTM-CNNs. *Transactions of the ACL*, 4:357–370, 2016.
- [17] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura. Intent-Based Networking - Concepts and Definitions. Internet-Draft draft-irtf-nmrg-ibn-concepts-definitions-00, Internet Engineering Task Force, December 2019. Work in Progress.
- [18] F. Chollet et. al. Keras, 2015. <https://github.com/fchollet/keras>.
- [19] N. Feamster and J. Rexford. Why (and How) Networks Should Run Themselves. In *Proceedings of the Applied Networking Research Workshop*, ANRW '18, page 20, New York, NY, USA, 2018. ACM.
- [20] J. R. Finkel, T. Grenager, and C. Manning. Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling. In *Proceedings of the 43rd Annual Meeting on ACL*, Acl '05, pages 363–370, Stroudsburg, PA, USA, 2005. ACL.
- [21] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. *SIGPLAN Not.*, 46(9):279–291, September 2011.

- [22] N. Foster, N. McKeown, J. Rexford, G. Parulkar, L. Peterson, and O. Sunay. Using Deep Programmability to Put Network Owners in Control. *SIGCOMM CCR*, 50(4):82–88, October 2020.
- [23] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-Driven Streaming Network Telemetry. In *Proceedings of the Annual Conference of the ACM SIGCOMM*, SIGCOMM '18, page 357–371, New York, NY, USA, 2018. ACM.
- [24] S. M. Harabagiu, A. Hickl, and V. F. Lacatusu. Negation, Contrast and Contradiction in Text Processing. In *AAAI*, 2006.
- [25] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Comput.*, 9(8):1735–1780, November 1997.
- [26] M. Honnibal and I. Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. *To appear*, 2017.
- [27] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks. In Rupak Majumdar and Viktor Kunčák, editors, *Computer Aided Verification*, pages 3–29, Cham, 2017. Springer International Publishing.
- [28] Google Inc. Dialogflow, March 2018. <https://dialogflow.com/>.
- [29] A. S. Jacobs, R. J. Pfitscher, R. A. Ferreira, and L. Z. Granville. Refining Network Intents for Self-Driving Networks. In *Proceedings of the Annual Conference of the ACM SIGCOMM Afternoon Workshop on Self-Driving Networks*, SelfDN 2018, pages 15–21, New York, NY, USA, 2018. ACM.
- [30] D. Jurafsky and J. H. Martin. *Speech and Language Processing*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 3rd edition, 2019.
- [31] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable Dynamic Network Control. In *12th USENIX Symposium on Networked Systems Design and Implementation*, pages 59–72, Berkeley, CA, USA, 2015. USENIX Association.
- [32] B. Koley. The Zero Touch Network, 2016. Keynote Speech at the 12th International Conference on Network and Service Management. <https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45687.pdf>.
- [33] J. D. Lafferty., A. McCallum, and F. C. N. Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the 18th International Conference on Machine Learning*, ICML '01, pages 282–289, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.
- [34] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer. Neural Architectures for Named Entity Recognition. In *Proceedings of the 2016 Conference of the North American Chapter of the ACL: Human Language Technologies*, pages 260–270, San Diego, California, June 2016. ACL.
- [35] B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.
- [36] N. Limsopatham and N. Collier. Bidirectional LSTM for Named Entity Recognition in Twitter Messages. In *Proceedings of the 2nd Workshop on Noisy User-generated Text*, NUTCOLING 2016, Osaka, Japan, December 11, 2016, pages 145–152, 2016.
- [37] V. Lingam, S. Bhuria, M. Nair, D. Gurpreetsingh, A. Goyal, and A. Sureka. Deep learning for conflicting statements detection in text. *PeerJ Preprints*, 6:e26589v1, March 2018.
- [38] H. H. Liu. The Practice of Network Verification in Alibaba's Global WAN, May 2021. <https://netverify.fun/the-practice-of-network-verification/-in-alibaba-global-wan/>.
- [39] Lumi. Lumi supplemental material, January 2020. <https://lumichatbot.github.io/>.
- [40] B. MacCartney, T. Grenager, M. C. de Marneffe, D. Cer, and C. D. Manning. Learning to Recognize Features of Valid Textual Entailments. In *Proceedings of the Main Conference of the North American Chapter of the ACL*, NAACL '06, pages 41–48, Stroudsburg, PA, USA, 2006. ACL.
- [41] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP Natural Language Processing Toolkit. In *ACL System Demonstrations*, pages 55–60, 2014.
- [42] M. C. De Marneffe, A. N. Rafferty, and C. D. Manning. Finding contradictions in text. In *Proceedings of the Conference of the ACL*, ACL '08, pages 1039–1047, 2008. cited By 108.
- [43] M. Marrero, J. Urbano, S. Sánchez-Cuadrado, J. Morato, and J. M. Gómez-Berbís. Named Entity Recognition: Fallacies, Challenges and Opportunities. *Computer Standards & Interfaces*, 35(5):482–489, 2013.

- [44] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient Estimation of Word Representations in Vector Space. *CoRR*, abs/1301.3781, 2013.
- [45] J. Mogul. Unsafe at Any Speed? Self-Driving Networks without Self-Crashing Networks. In *Keynote Speech at the ACM SIGCOMM Afternoon Workshop on Self-Driving Networks*, SelfDN 2018, New York, NY, USA, 2018. ACM.
- [46] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 1–13, Lombard, IL, 2013. USENIX Association.
- [47] Juniper Networks. What Is Intent-Based Networking? <https://www.juniper.net/us/en/products-services/what-is/intent-based-networking/>.
- [48] OpenConfig. OpenConfig, January 2016. <http://www.openconfig.net/>.
- [49] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. <http://scikit-learn.org>.
- [50] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, Santa Clara, CA, February 2020. USENIX Association.
- [51] C. Prakash, J. Lee, Y. Turner, J. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *Proceedings of the Annual Conference of the ACM SIGCOMM*, SIGCOMM '15, pages 29–42, New York, NY, USA, 2015. ACM.
- [52] Microsoft Research. Hyperscale cloud reliability and the art of organic collaboration, Nov 2018. <https://www.microsoft.com/en-us/research/blog/hyperscale-cloud-reliability-and-the-art-of-organic-collaboration>.
- [53] A. Ritter, D. Downey, S. Soderland, and O. Etzioni. It's a Contradiction—no, It's Not: A Case Study Using Functional Relations. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, Emnlp '08, pages 11–20, Stroudsburg, PA, USA, 2008. ACL.
- [54] L. Ryzhyk, N. Bjørner, M. Canini, J. Jeannin, C. Schlesinger, D. B. Terry, and G. Varghese. Correct by Construction Networks Using Stepwise Refinement. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 683–698, Boston, MA, 2017. USENIX Association.
- [55] F. Sarafraz. *Finding conflicting statements in the biomedical literature*. PhD thesis, University of Manchester, UK, 2012.
- [56] E. J. Scheid, C. C. Machado, M. F. Franco, R. L. dos Santos, R. P. Pfitscher, A. E. Schaeffer-Filho, and L. Z. Granville. INSpIRE: Integrated NFV-based Intent Refinement Environment. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 186–194, May 2017.
- [57] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A Language for Managing Network Resources. *IEEE/ACM Transactions on Networking*, 26(5):2188–2201, October 2018.
- [58] Y. E. Sung, C. Lund, M. Lyn, S. G. Rao, and S. Sen. Modeling and Understanding End-to-End Class of Service Policies in Operational Networks. In *Proceedings of the Annual Conference of the ACM SIGCOMM*, SIGCOMM '09, page 219–230, New York, NY, USA, 2009. ACM.
- [59] Y. E. Sung, X. Tie, S. H. Y. Wong, and H. Zeng. Robotron: Top-down Network Management at Facebook Scale. In *Proceedings of the Annual Conference of the ACM SIGCOMM*, SIGCOMM '16, pages 426–439, New York, NY, USA, 2016. ACM.
- [60] N. S. Tawfik and M. R. Spruit. Automated Contradiction Detection in Biomedical Literature. In Petra Perner, editor, *Machine Learning and Data Mining in Pattern Recognition*, pages 138–148, Cham, 2018. Springer International Publishing.
- [61] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the ACL*, pages 252–259, 2003.
- [62] VMware. Intent-based Networking, May 2021. <https://www.vmware.com/topics/glossary/content/intent-based-networking>.
- [63] V. Yadav and S. Bethar. A Survey on Recent Advances in Named Entity Recognition from Deep Learning model. In *Proceedings of the 27th International Conference on Computational Linguistic*, pages 2145–215. ACL, 2018.

Boosting Full-Node Repair in Erasure-Coded Storage

Shiyao Lin¹, Guowen Gong¹, Zhirong Shen^{1*}, Patrick P. C. Lee², and Jiwu Shu^{1,3}
¹*Xiamen University* ²*The Chinese University of Hong Kong* ³*Tsinghua University*

Abstract

As a common choice for fault tolerance in today’s storage systems, erasure coding is still hampered by the induced substantial traffic in repair. A variety of erasure codes and repair algorithms are designed in recent years to relieve the repair traffic, yet we unveil via careful analysis that they are still plagued by several limitations, which restrict or even negate the performance gains. We present RepairBoost, a scheduling framework that can assist existing linear erasure codes and repair algorithms to boost the full-node repair performance. RepairBoost builds on three design primitives: (i) repair abstraction, which employs a directed acyclic graph to characterize a single-chunk repair process; (ii) repair traffic balancing, which balances the upload and download repair traffic simultaneously; and (iii) transmission scheduling, which carefully dispatches the requested chunks to saturate the most unoccupied bandwidth. Extensive experiments on Amazon EC2 show that RepairBoost can accelerate the repair by 35.0-97.1% for various erasure codes and repair algorithms.

1 Introduction

Today’s storage systems are often composed of a large number of *storage nodes* (called “nodes” for brevity) to accommodate the explosively increasing data volume, making failures arise unexpectedly yet prevalently. To protect data reliability even in the presence of failures, many storage systems resort to *replication* [14] and *erasure coding* [47], both of which rely on pre-storing additional redundancy to repair the lost data. As opposed to replication that simply stores identical replicas, erasure coding can assuredly attain the same fault tolerance degree with much less storage consumption [59], and hence is much more preferable in commodity storage systems [4–6, 41, 43]. In principle, erasure coding consists of two lightweight computational operations, namely *encoding* (i.e., generating redundant chunks based on the original data chunks) and *decoding* (i.e., repairing the original data chunks based on the

surviving chunks), so as to realize the efficient transformation between the original data and redundancy.

Although being storage-efficient, erasure coding is prone to substantial *repair traffic* (i.e., the amount of data transmitted over the network for repair), as it often needs to retrieve multiple surviving chunks to repair a single chunk. To relieve the I/O amplification problem in repair, existing studies mainly resort to the following approaches: (i) constructing new theoretical erasure codes with provably reduced repair traffic (e.g., Locally Repairable Codes [21, 28, 52], Rotated-RS codes [27], and regenerating codes [13, 44, 49, 58]); (ii) designing efficient repair algorithms to parallelize the repair process [22, 32, 40, 56]; and (iii) utilizing machine-learning-based prediction techniques [16, 34, 42] to proactively restore the data with the repair algorithms before failure occurrence [38, 54] (see §2.2 for details).

Our observation is that most existing erasure codes and repair algorithms mainly focus on the single-chunk repair, yet the full-node failure (i.e., all the chunks in a node are permanently lost) must simultaneously manipulate the repair of multiple chunks. Thus, *there exists a gap between the deployment of existing repair approaches and the requirement of full-node repair*. Such a gap leads to several practical limitations: (i) they do not specifically leverage the full-duplex transmission to saturate the available bandwidth; (ii) they fail to carefully schedule the transmission of chunks to fully utilize the bandwidth at all times; (iii) they neglect the elastic cooperation of different repair algorithms to meet diverse reliability guarantees [25] and access popularities [9, 23]; and (iv) they have dedicated repair strategies with the pre-specified data routing among nodes, thereby increasing the implementation complexity (see §2.3 for details). Therefore, how to seamlessly deploy existing repair approaches to efficiently tackle the full-node repair remains a challenging yet crucial issue in erasure-coded storage.

We bridge this gap by designing RepairBoost, a scheduling framework that can assist a variety of erasure codes and repair algorithms to speed up a full-node repair. The main idea behind RepairBoost is to formulate a single-chunk re-

*Corresponding author: Zhirong Shen (shenzr@xmu.edu.cn).

pair through a *repair directed acyclic graph* (RDAG), which characterizes the data routing over the network and the dependencies among the requested chunks for repair. RepairBoost then decomposes an RDAG into multiple *repair tasks*, each of which performs data uploads and downloads to facilitate the repair. RepairBoost balances the repair traffic and bandwidth utilization in the full-node repair through the following two steps: (i) it carefully dispatches the repair tasks of multiple RDAGs to the corresponding nodes for balancing the overall upload and download repair traffic; and (ii) it coordinates the execution orders of repair tasks to saturate the utilization of the available upload and download bandwidth. To summarize, our main contributions include:

- **[Design]** We propose RepairBoost to assist existing erasure codes and repair algorithms for speeding up the full-node repair. RepairBoost formulates a single-chunk repair through an RDAG. It then balances the upload and download repair traffic via carefully assigning the repair tasks of RDAGs to the nodes. RepairBoost also formulates the maximum flow problem [57] to schedule the data transmission for saturating the unoccupied bandwidths (§3.1-§3.3).
- **[Generality]** We also show that RepairBoost can be extended to tackle multiple node failures and boost the repair in heterogeneous environments (§3.4).
- **[Implementation]** We implement a prototype of RepairBoost with C++, which can be an independent middleware deployed atop existing storage systems for repair scheduling. We also demonstrate the portability of RepairBoost by integrating it into Hadoop HDFS 3.1.4 with limited modifications to the codebase (with around 270 LoC added) (§4).
- **[Evaluation]** We evaluate the performance of RepairBoost on Amazon EC2 [8], showing that it can support a variety of erasure codes and repair algorithms and increase the repair throughput by 35.0-97.1% (§5).

The source code of our RepairBoost prototype can be reached via <https://github.com/shenzr/repairboost-code>.

2 Background

We start with the basics of erasure coding (§2.1) and elaborate on existing attempts for repair acceleration in erasure-coded storage (§2.2). We also summarize the limitations that remain to be addressed (§2.3).

2.1 Basics

Erasur coding introduces slight computational operations to reduce the storage overhead for reliability assurance [59]. It operates on data in units of *chunks*, which are a collection of data with the size of several megabytes (e.g., 64 MB in Hadoop HDFS [6] and 256 MB in Facebook’s data-warehouse cluster [48]). Formally, erasure coding can be con-

figured via two integer parameters, namely k and m , which tune the storage efficiency and fault tolerance assurance. A (k, m) erasure code encodes every k equal-sized data chunks $\{D_1, D_2, \dots, D_k\}$ at a time to generate additional m redundant chunks (called *parity chunks*) $\{P_{k+1}, P_{k+2}, \dots, P_{k+m}\}$. In this paper, we mainly consider the *linear erasure codes*, where each parity chunk can be expressed as a *linear combination* of the k data chunks via the Galois Field arithmetic [46], given by $P_{k+j} = \sum_{i=1}^k \alpha_{i,j} D_i$, where $\alpha_{i,j}$ ($1 \leq i \leq k$ and $1 \leq j \leq m$) is the encoding coefficient used by the data chunk D_i to calculate the parity chunk P_{k+j} . Such $k+m$ data and parity chunks that are encoded together collectively constitute a *stripe* (or a coding group [33, 62]), ensuring that *any* k out of $k+m$ chunks of the same stripe *always* suffice to restore (decode) the original k data chunks (a.k.a. *Maximum Distance Separable* (MDS) property in coding theory). In other words, the (k, m) erasure code can tolerate *any* m chunk failures for every stripe.

In practice, data chunks in storage systems are often organized into multiple stripes, which are *independently* manipulated by erasure coding. Hence, by distributing the $k+m$ chunks of each stripe across $k+m$ distinct nodes (i.e., one chunk per node), we can always guarantee data reliability in the face of no more than *any* m node failures. In this paper, we equally treat the data and parity chunks in repair, and just refer to them as “chunks.”

A variety of erasure codes have been proposed for decades, where Reed-Solomon (RS) codes [51] are the most popular code construction used in production storage (e.g., Ceph [4], Hadoop HDFS [6], QFS [43], Facebook f4 [41], and Yahoo Cloud Object Store [5]), since they support any parameters of k and m . For simplicity, we denote the RS code configured by the parameters k and m as $RS(k, m)$ and use it as our case study throughout the paper. For example, Figure 1(a) depicts a stripe of $RS(4, 2)$. We also show that our work is applicable to other linear erasure codes (§5.3).

2.2 Repair

Repair in erasure-coded storage is often classified into (i) *full-node repair*, which restores all lost chunks in a failed node (e.g., caused by a disk failure), and (ii) *degraded reads* to the temporarily unavailable chunks (e.g., caused by system upgrades and network disconnections) or the chunks that have not yet been repaired in the full-node repair. In this paper, we mainly focus on the *single full-node repair*, which is recognized as one of the top causes of service downtime [35]. Our work can also tackle multiple node failures (§3.4).

The full-node repair has to manipulate the repair across multiple stripes. Although being storage-efficient, erasure coding is prone to a high repair penalty. For example, $RS(k, m)$ needs to retrieve k surviving chunks of the same stripe to recover a lost chunk, thereby amplifying the storage and network I/Os in repair by k times. Specifically, suppose that a chunk C^* fails and $\{C_1, C_2, \dots, C_k\}$ is the set of k chunks

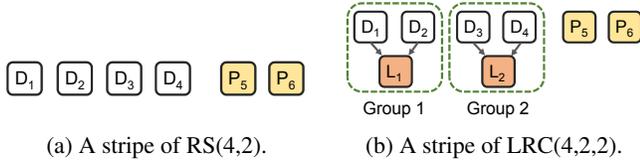


Figure 1: Examples of RS codes and LRCs.

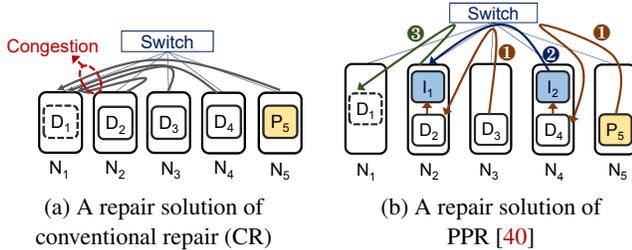


Figure 2: Repair solutions for RS(4,2). D_1 indicates the repaired chunk. N_1 is the destination node.

within the same stripe of C^* . We can repair C^* using a linear combination of the k chunks as:

$$C^* = \sum_{i=1}^k \beta_i C_i, \quad (1)$$

where β_i is the decoding coefficient used by C_i for repairing C^* . In this paper, we refer to the node storing the repaired chunk as the *destination node*.

In view of the high repair penalty, existing studies design solutions that mitigate the repair penalty as follows.

Repair-efficient erasure codes: They relieve the repair traffic with theoretical guarantees through the construction of erasure codes, where *Locally Repairable Codes* (LRCs) [21, 52] and regenerating codes [13, 44, 48, 49, 58] are two representatives. LRCs are configured via three parameters, namely k , l , and m . $LRC(k, l, m)$ extends $RS(k, m)$ by organizing the k data chunks of a stripe into l groups (assuming that k is divisible by l) and maintaining an additional *local parity chunk* for each group. This design allows a single data chunk to be repairable by retrieving merely $\frac{k}{l}$ surviving chunks within the same group¹. Figure 1(b) shows a stripe of $LRC(4, 2, 2)$, where L_i is the local parity chunk of the i -th group ($1 \leq i \leq 2$).

On the other hand, regenerating codes reduce the repair traffic by (i) requiring surviving nodes to send the linear combination of the locally stored data [13] and (ii) contacting more surviving nodes to assist the repair [49]. Recently proposed regenerating codes (e.g., Butterfly codes [44] and Clay codes [58]) further eliminate the needs of computations in the surviving nodes, which can simply retrieve the required sub-chunks directly from surviving nodes for repair.

Repair algorithms: While erasure codes specify *which chunks* should be retrieved for repair, repair algorithms mainly focus on *how* to quickly retrieve the surviving chunks over

the network for existing erasure codes. They accelerate the repair by exploiting the unoccupied bandwidth without reducing the repair traffic. Each repair algorithm specifies a *repair solution* for a lost chunk, including the data routing and execution orders among the surviving nodes. Figure 2(a) shows the repair solution of D_1 under the *conventional repair* (CR) for $RS(4, 2)$, where it transmits four surviving chunks $\{D_2, D_3, D_4, P_5\}$ at the same time to the destination node N_1 . Suppose that a chunk can be transmitted over a network link in a *timeslot*. The conventional repair takes four timeslots for N_1 to download the four chunks, because of the congestion of the download link of N_1 . PPR [40] relieves the network congestion by decomposing a single-chunk repair into multiple sub-stages and executing them in parallel, so as to fully utilize the available bandwidth. Figure 2(b) depicts a single-chunk repair solution under PPR for $RS(4, 2)$, which accomplishes the repair in three timeslots: (i) in the first timeslot (1), we transmit D_3 to N_2 and add them together (as in Equation (1)) to generate an intermediate chunk I_1 , while at the same time sending P_5 to N_4 to generate another intermediate chunk I_2 ; (ii) in the second timeslot (2), we transmit I_2 to N_1 and add I_2 with I_1 to restore the lost chunk D_1 (as in Equation (1)); and (iii) in the third timeslot (3), we deliver D_1 to the destination node. Thus, PPR exploits the available bandwidth across multiple nodes at each timeslot to mitigate the network congestion in a single-chunk repair. ECPipe [32] further reduces the repair time to almost one timeslot by decomposing the repair of a lost chunk into the pipelined repair operations of multiple smaller sub-chunks.

Proactive repair with erasure coding: Both repair-efficient erasure codes and repair algorithms are *reactive*, implying that they launch the repair operation only *after* failures truly occur. *Proactive* repair [38, 54] further reduces the window of data vulnerability by using *machine-learning-based failure prediction models* (e.g., Bayes classifier [16], support vector machines [42], and random forest [34]), such that it can detect in advance the nodes impending to fail and *proactively* repair the data in such nodes *before* the failure occurrence. Proactive repair can be realized by the data migration and the decoding of chunks based on erasure coding [54], where we mainly focus on the latter in this paper.

2.3 Limitations

By examining existing attempts to accelerate the repair of erasure-coded storage, we identify the following limitations that, if not properly addressed, will limit the performance gains when they are directly employed for the full-node repair.

Limitation 1 (L1): Failing to utilize the full duplex transmission. Most existing studies overlook the *full duplex transmission* [12, 36] in repair, which enables a node to send (upload) and receive (download) data simultaneously and independently; as a result, they cannot properly balance the upload and download repair traffic in the full-node repair. Figure 3 shows two examples of repair under CR (where N_i is the i -th

¹LRC(k, l, m) also needs k chunks to repair a global parity chunk (e.g., P_5 and P_6 in Figure 1(b)).

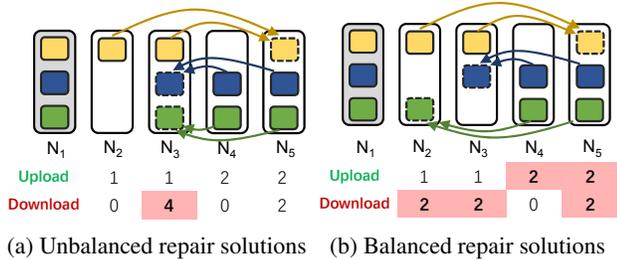


Figure 3: L1 (Failing to utilize the full duplex transmission). The repair time is determined by the most loaded node (with the red color in the background).

node), where the first example needs to download four chunks (at N_3) to accomplish the repair, while the latter only uploads and downloads at most two chunks for repair. This example indicates that balancing the upload and download repair traffic in the full duplex transmission has the potential to reduce the overall repair time, which is determined by the node with the most upload or download repair traffic.

Specifically, the repair-efficient codes [13, 21, 44, 48, 49, 58] reduce a single chunk’s repair traffic without concerning traffic balancing. Some repair algorithms (e.g., ECPipe [32] and PPR [40]) mainly focus on accelerating a single chunk’s repair, while paying limited attention to balance the upload and download repair traffic in the full-node repair. Although ClusterSR [55] balances the inter-cluster upload and download repair traffic, it still does not balance the upload and download bandwidths in general storage systems.

Limitation 2 (L2): Failing to fully utilize the bandwidth at each timeslot. While existing repair algorithms can relieve the download bottleneck within a single-chunk repair (e.g., PPR [40] and ECPipe [32]), they simply combine the repair solutions of multiple chunks to cope with the full-node repair. This may unintentionally lead to the link congestion again and make the bandwidth under-utilized in the full-node repair.

Figure 4 shows how transmission scheduling affects the repair. In Figure 4(a), after the transmissions of C_2 and C_4 in the first timeslot, the transmissions of C_3 (from the node N_5 to N_3) and C_7 (from N_2 to N_3) compete for the bandwidth of the download link of N_3 . Figure 4(a) gives priority to C_3 in the second timeslot. Since C_6 can only be transmitted after receiving C_7 , Figure 4(a) finally needs four timeslots. As a comparison, Figure 4(b) sends C_7 at the second timeslot, allowing the transmission of C_6 and C_3 to be performed in parallel without bandwidth competition. Hence, Figure 4(b) only uses three timeslots. This example indicates that the repair solutions in the full-node repair should be carefully scheduled.

Limitation 3 (L3): Inflexibility. Many repair algorithms [22, 32, 40, 55, 56] treat every chunk equally and repair all the lost chunks using the same repair algorithm. This repair fashion is simple yet inflexible, as it cannot elastically combine different repair algorithms, and make them cooperate for diverse reliability requirements [25] and skewed access popu-

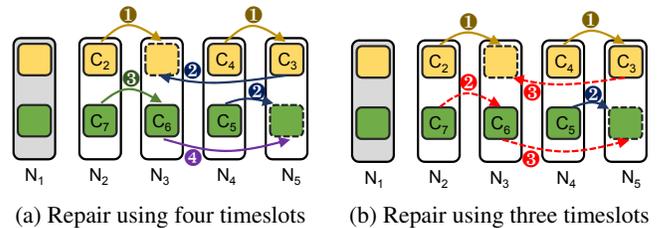


Figure 4: L2 (Failing to fully utilize the bandwidth at each timeslot). The transmission scheduling affects the bandwidth utilization.

larity in real-world storage systems [9, 23]. For example, we can combine ECPipe [32] and CR, such that we use ECPipe to repair the chunks that require higher reliability guarantee, while employing CR to restore the remaining ones, so as to regulate the repair traffic and foreground traffic.

Limitation 4 (L4): Lack of a general framework for the full-node repair. Existing distributed storage systems deploy different specific repair algorithms for a single-chunk repair. For example, many commodity storage systems (e.g., Hadoop HDFS [7] and Windows Azure Storage [21]) still rely on CR in a single-chunk repair due to its simplicity, while PPR [40] and CAR [56] can be used in hierarchical storage systems to reduce inter-rack repair traffic. It is desirable to have a *general* framework that simultaneously supports different types of repair algorithms for different deployment scenarios.

3 RepairBoost Design

We present RepairBoost, a scheduling framework that can assist a variety of erasure codes and repair algorithms to boost the full-node repair performance.

Assumptions: RepairBoost is designed based on the following assumptions. First, RepairBoost mainly focuses on a single full-node repair, which is reported as the most predominant failure event in practice (e.g., 98% of the total failure events [50]). Nevertheless, RepairBoost can be extended to tackle a multi-node failure (i.e., more than one node in a stripe fails) (§3.4). Second, to simplify our discussion, we elaborate RepairBoost using RS codes in homogeneous environments (i.e., with identical link bandwidth), yet we also show that RepairBoost works for other erasure codes (§5.3) and can be deployed in heterogeneous environments (i.e., with different link bandwidth) (§3.4).

Overview: RepairBoost uses the following techniques to address the limitations aforementioned (§2.3). It first abstracts a single-chunk repair solution via a general *repair directed acyclic graph (RDAG)* (§3.1). By supporting the scheduling of multiple RDAGs, RepairBoost can achieve both *flexibility* (i.e., allowing the collaboration of different repair algorithms; L3 addressed) and *generality* (i.e., being workable for various erasure codes and repair algorithms; L4 addressed).

RepairBoost then decomposes multiple RDAGs into vertices that have dedicated repair tasks. It then carefully assigns

the repair tasks to the nodes, so as to balance the overall upload and download repair traffic across the surviving nodes (§3.2; L1 addressed).

RepairBoost finally constructs a directed network based on the surviving nodes and the corresponding repair tasks. It then determines the chunks to be transmitted by solving a maximum flow problem [57], so as to fully saturate the unoccupied upload and download bandwidth at each timeslot (§3.3; L2 addressed).

3.1 Repair Abstraction

RDAG construction: We first formalize a single-chunk repair solution through a *directed acyclic graph* (DAG), which is called the *Repair DAG* (RDAG). For $RS(k, m)$, the RDAG of a lost chunk can be initialized over $k + 1$ vertices, where the k vertices $\{v_1, v_2, \dots, v_k\}$ represent the k nodes that store the requested surviving chunks for repair, while v_{k+1} denotes the destination node. Also, we employ directed edges among vertices to represent the data routing directions specified in repair algorithms.

We construct the edges based on the following rules. Given two vertices v_i and v_j ($1 \leq i \neq j \leq k + 1$), we use a directed edge $e_{i,j}$ to indicate that v_i is designated to send a surviving chunk to v_j . Hence, if $e_{i,j}$ exists, we say that v_i is a *child* of v_j , and conversely v_j is the *parent* of v_i . For v_j (where $j \neq k + 1$), it has to collect all the requested surviving chunks from its children, add them together with the local data it stores using the decoding coefficient (see Equation (1)), and send the result to its parent. Therefore, in this RDAG, v_j can send a chunk for repair once all the chunks required from its children are received. As the full-node repair has to recover multiple chunks, a node may have multiple parents and children across different RDAGs.

Repair process guided by RDAG: The repair starts from the *leaf vertices* (i.e., those that do not have any child) and ends at v_{k+1} : for each edge $e_{i,j}$ ($1 \leq i \neq j \leq k + 1$), if v_i has sent the requested chunk to v_j for repair, we remove $e_{i,j}$ from the RDAG; further, for each leaf vertex v_i , if there is no edge connecting to it (indicating that v_i has already transmitted all the requested chunks to its parents), we remove v_i from the RDAG as well. Hence, as the repair proceeds, the number of vertices in the RDAG will gradually decrease and the lost chunk is successfully repaired once the vertex v_{k+1} becomes a leaf vertex eventually.

Example: We take the RDAG of PPR in Figure 5 as an example, where $k = 4$. Suppose that the four requested surviving chunks in $\{v_1, v_2, \dots, v_4\}$ are denoted by $\{C_1, C_2, \dots, C_4\}$, respectively. In the first stage, v_1 sends a chunk $\beta_1 C_1$ to its parent v_2 (where β_1 is the decoding coefficient of C_1 ; see Equation (1)), while at the same time v_3 sends another chunk $\beta_3 C_3$ to its parent v_4 . We also update the RDAG by removing $e_{1,2}$, $e_{3,4}$, v_1 , and v_3 . In the second stage, the leaf vertex v_2 combines $\beta_1 C_1$ with its stored chunk C_2 and sends the result $\beta_1 C_1 + \beta_2 C_2$ to its parent v_4 . In the third stage, after collect-

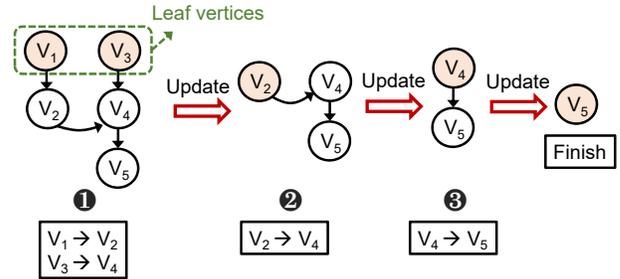


Figure 5: Example of an RDAG of PPR [40] when $k = 4$. The vertex with pink color indicates the leaf vertex. We repeatedly update an RDAG as the repair proceeds.

ing two chunks from its children, v_4 combines $\beta_1 C_1 + \beta_2 C_2$, $\beta_3 C_3$, and its locally stored chunk C_4 to restore the lost chunk $C^* = \sum_{i=1}^4 \beta_i C_i$, and finally sends C^* to v_5 . The repair of C^* completes when v_5 becomes a leaf vertex.

Advantages of an RDAG: The RDAG design has the following advantages. First, an RDAG is a general formalization of a single-chunk repair solution. That is, we can construct an RDAG for any given repair algorithm once we identify the surviving chunks needed for repair and the data routing strategy performed among them. Thus, RepairBoost is applicable to a variety of erasure codes (e.g., RS codes, LRCs, and regenerating codes) and single-stripe repair algorithms (e.g., CR, PPR, and ECPipe). This design also resolves the tensions of deploying specific repair algorithms (i.e., being general) and facilitates their co-existence (i.e., being flexible).

Second, an RDAG describes how data is transmitted over the network and also the dependency (or relationship) among the k surviving chunks involved in the repair. Given an RDAG, we can readily identify that the leaf vertices with different parents in an RDAG can be potentially transmitted in parallel to occupy the available bandwidth without network congestion (e.g., v_1 and v_3 in Figure 5). We leverage this property in the transmission scheduling (§3.3).

Third, an RDAG also indicates the repair tasks of each vertex. For example, we can learn that v_4 in Figure 5 needs to download two chunks and upload one chunk in the repair. We explore the traffic balancing through the assignment of repair tasks (§3.2).

Discussion: OpenEC [29] proposes an ECDAG to characterize the encoding and decoding processes. Both an RDAG and an ECDAG are fundamentally different in the following aspects: (i) *physical meanings of edges*: the edges in an ECDAG represent the encoding and decoding operations, while the edges in an RDAG specify the routing and dependencies in repair; (ii) *graph structures*: an ECDAG introduces *virtual vertices* to denote the intermediate chunks generated, while an RDAG is built over the chunk repaired and the surviving chunks for repair; and (iii) *graph maintenance*: an ECDAG keeps the graph unchanged throughout the encoding and decoding processes, while an RDAG is iteratively updated with the repair progress.

Algorithm 1 Mapping of Intermediate Vertices

Input: \mathcal{T} (set of intermediate vertices)**Output:** The mapping \mathcal{M} from intermediate vertices to nodes

```
1: procedure MAIN( $\mathcal{T}$ )
2:   Set  $\mathcal{M} = \emptyset$ 
3:   Sort  $\mathcal{T}$  in descending order
4:   while  $\mathcal{T} \neq \emptyset$  do
5:      $\bar{v} = \text{POP}(\mathcal{T})$ 
6:     Establish  $\mathcal{N}$ 
7:      $N^* = \text{MAP}(\bar{v}, \mathcal{N})$ 
8:     Append  $(\bar{v}, N^*)$  to  $\mathcal{M}$ 
9:   end while
10:  return  $\mathcal{M}$ 
11: end procedure
12: function MAP( $\bar{v}, \mathcal{N}$ )
13:  Set  $N^* = \arg \min\{d_{N_i} | N_i \in \mathcal{N}\}$ 
14:   $\mathcal{T} = \mathcal{T} - \bar{v}$ 
15:   $u_{N^*} = u_{N^*} + u_{\bar{v}}$ 
16:   $d_{N^*} = d_{N^*} + d_{\bar{v}}$ 
17:  return  $N^*$ 
18: end function
```

3.2 Repair Traffic Balancing

After constructing the RDAGs for the lost chunks in a failed node, RepairBoost then assigns the repair tasks by mapping the vertices to the nodes, such that (i) the fault tolerance degree (i.e., the tolerable number of failed nodes) offered by erasure coding must be preserved after repair, and (ii) the upload and download repair traffic across the whole system should be as balanced as possible.

Retaining fault tolerance degree: Given an RDAG of a lost chunk, we assign the k vertices $\{v_1, v_2, \dots, v_k\}$ to the k nodes that store the surviving chunks of the same stripe. We also map v_{k+1} to the node that does not store any chunk of the same stripe before the repair. By doing so, RepairBoost ensures that the $k + m$ chunks of the same stripe still reside in $k + m$ different nodes after repair, thereby retaining the node-level fault tolerance offered by erasure coding (§2.1).

Balancing repair traffic: Given an RDAG, we represent the repair task of a vertex v_i by a tuple (u_{v_i}, d_{v_i}) (where $1 \leq i \leq k + 1$), which indicates the numbers of chunks being uploaded and downloaded by v_i in this RDAG, respectively.

We classify the vertices of an RDAG into three categories: (i) the leaf vertex v_i , which only sends (uploads) surviving chunks for repair (i.e., $u_{v_i} > 0$ and $d_{v_i} = 0$); (ii) the *root vertex*, which only receives (downloads) data for repair (i.e., $u_{v_i} = 0$ and $d_{v_i} > 0$); and (iii) the intermediate vertex, which receives (downloads) multiple chunks from its children and sends (uploads) one intermediate chunk to its parent (i.e., $u_{v_i} = 1$ and $d_{v_i} > 0$). For example, in Figure 5, v_1 is a leaf vertex, v_2 is an intermediate vertex, and v_5 is the root vertex.

After collecting the three kinds of vertices from multiple RDAGs, our main idea is to give priority to map the intermediate and root vertices to the nodes, with the primary objective

of balancing the download repair traffic at first. We then carefully assign the leaf vertices to further balance the upload repair traffic. Algorithm 1 shows the pseudo-code of the mapping algorithm for the intermediate vertices.

Algorithm details: Let \mathcal{T} be the set of the intermediate vertices that have not been assigned yet. Let \mathcal{M} be the mapping established. RepairBoost first initializes \mathcal{M} as an empty set and sorts the intermediate vertices in \mathcal{T} by the number of chunks required to be downloaded in descending order (Lines 2-3 in Algorithm 1). It then pinpoints the vertex \bar{v} that needs to download the most chunks for repair among the vertices of \mathcal{T} (Line 5). RepairBoost finds the set of nodes (denoted by \mathcal{N}) that store the surviving chunks requested in the RDAG of \bar{v} but have not been assigned the repair task of this RDAG (Line 6). It then searches the node for \bar{v} by calling the MAP function (Line 7).

In the MAP function, RepairBoost selects the node N^* that has the least download traffic among the ones in \mathcal{N} (Line 13), where d_{N_i} denotes the number of downloaded chunks of the node N_i . It then maps \bar{v} to N^* . RepairBoost later excludes \bar{v} from the set of \mathcal{T} (Line 14), and increases the numbers of the uploaded and downloaded chunks (denoted by u_{N^*} and d_{N^*}) of N^* by the task of \bar{v} (Lines 15-16). The mapping indicates that N^* will serve as \bar{v} in the corresponding RDAG by uploading and downloading the requested surviving chunks specified to \bar{v} . RepairBoost repeats the above steps until all the intermediate vertices have been assigned to the corresponding nodes (Lines 4-9).

We analyze the computational complexity of Algorithm 1. Suppose that the number of nodes participating in repair is n and $|\mathcal{T}|$ is the number of intermediate vertices in \mathcal{T} . We note that: (i) the sorting of the intermediate vertices (Line 3) incurs a computational complexity of $O(|\mathcal{T}| \cdot \log(|\mathcal{T}|))$, and (ii) the mapping of an intermediate vertex (Lines 5-8) incurs a computational complexity of $O(n \cdot \log n)$, which will be executed for $|\mathcal{T}|$ times (Line 4). Thus, the overall computational complexity of Algorithm 1 is $O(|\mathcal{T}| \cdot \log(|\mathcal{T}|) + |\mathcal{T}| \cdot n \cdot \log n)$.

RepairBoost then maps the root and leaf vertices to the corresponding nodes in the similar way, except the following differences: (i) when mapping the root vertex of an RDAG, RepairBoost selects the node with the lightest download repair traffic among the ones, which does not store any chunk within the same stripe of the repaired chunk; (ii) when mapping the leaf vertices of an RDAG, RepairBoost chooses the nodes with the lightest upload repair traffic among the ones, which not only store the surviving chunks within the same stripe of the repaired chunk, but also have not been mapped with any vertex of this RDAG.

Example: Figure 6 shows an example of mapping an RDAG to the nodes. Given an RDAG, we decompose it into vertices with the tuples (u_{v_i}, d_{v_i}) ($1 \leq i \leq k + 1$ and $k = 4$ in this example), which specify the numbers of uploaded and downloaded chunks for repair (Step 1). For example, v_2 needs to upload and download one chunk. To map the intermedi-

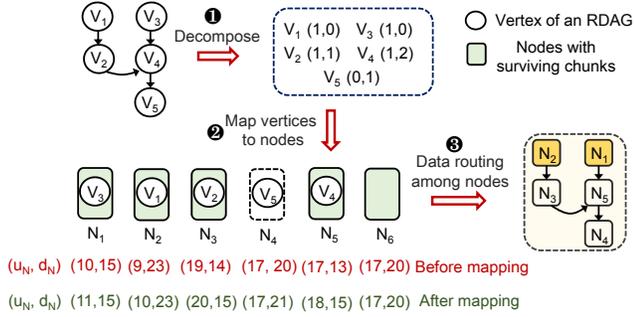


Figure 6: Example of mapping vertices of an RDAG to nodes. We select an RDAG of PPR to repair a chunk of RS(4, 2). N_2 and N_1 are the leaf vertices of this RDAG after mapping. The numbers in red (resp. green) denotes the upload and download traffics afforded by a node before (resp. after) the mapping.

ate vertices (i.e., v_2 and v_4), we first consider v_4 as it needs to download the most chunks. Since N_5 needs to download the fewest chunks (i.e., 13) among $\{N_1, N_2, N_3, N_5, N_6\}$ that store the surviving chunks, we map v_4 to N_5 , and update the numbers of uploaded and downloaded chunks afforded by N_5 afterwards (Step 2). After all the five vertices of an RDAG have been assigned, we can learn the data routings among nodes $\{N_1, N_2, \dots, N_5\}$ to repair the chunk (Step 3).

3.3 Transmission Scheduling

After establishing the mapping from the RDAGs to the nodes for balancing the overall upload and download repair traffic (§3.2), it does not necessarily achieve the lower-bound of the repair time, as the bandwidth may not be utilized at each timeslot during the repair (L2 in §2.3).

To further saturate the bandwidth utilization, RepairBoost formulates the transmission scheduling as a *maximum flow problem* [57]. Specifically, suppose that there are n nodes participating in the full-node repair. We can construct a network based on the RDAGs of the lost chunks. This network is built over $2n + 2$ vertices (Figure 7), with a source s , a sink t , n sender vertices $\{S_1, S_2, \dots, S_n\}$ representing the n nodes that can potentially send data for repair, and another n receiver vertices $\{R_1, R_2, \dots, R_n\}$ representing the n nodes that may receive data at the same time. We then establish the connections as follows: for any two vertices S_i and R_j ($1 \leq i \neq j \leq n$), we establish the connection between S_i and R_j once S_i can send a chunk to R_j according to the RDAGs. Each connection between S_i and R_j is assigned with the capacity of one, implying that we can send one surviving chunk at a time from S_i and R_j . Thus, our objective is to find a maximum flow over the network whose capacity denotes the most chunks that can be transmitted simultaneously at this timeslot, so as to saturate the available upload and download bandwidth.

After establishing the maximum flow, we dispatch the chunks according to the selected edges of the maximum flow. If S_i has many chunks to be sent, we prefer to send a chunk, such that sending this chunk can make a parent of S_i become a leaf vertex in an RDAG in the next timeslot (§3.1). Our

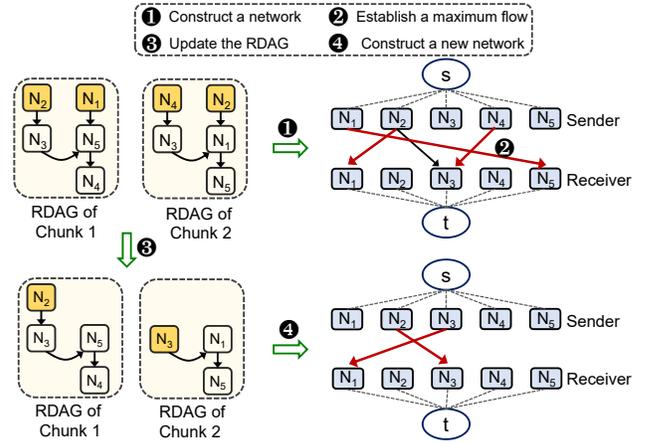


Figure 7: Example of transmission scheduling (where $k = 4$ and $n = 5$). The red arrows means the maximum flows established.

objective is to establish the maximum flow, while at the same time helping increase the number of edges of the network in the next scheduling. This can potentially increase the capacity of the maximum flow in the next transmission.

Once the chunks associated with the edges of the maximum flow have all been transmitted, we accordingly delete the corresponding edges in the RDAGs and update the network based on the residual RDAGs. We repeat the scheduling until all the lost chunks of the failed node have been repaired.

Example: Figure 7 shows an example to repair two chunks among five surviving nodes (i.e., $n = 5$). There are three nodes (i.e., N_1, N_2 , and N_4) that can send chunks for repair (marked in yellow color) in the RDAGs at the very beginning. We construct the network (Step 1) based on the two RDAGs and find the maximum flow (Step 2). The maximum flow indicates that we can send the chunks as follows for transmitting the most chunks simultaneously: $N_1 \rightarrow N_5$ (in the RDAG of the chunk 1), $N_2 \rightarrow N_1$ (in the RDAG of the chunk 2), and $N_4 \rightarrow N_3$ (in the RDAG of the chunk 2). We then update the RDAGs by removing the edges associated with the transmitted chunks and marking the leaf vertex at this time (Step 3). We repeat the construction of the network and the finding of the maximum flow for the residual RDAGs (Step 4).

Complexity analysis: Suppose that there are n nodes and e edges in a network. We can use Dinic’s algorithm [24] to find the maximum flow, whose computational complexity is $O(n^2e)$.

3.4 Extensions

Multi-node repair: We offer two options when using RepairBoost to cope with multi-node repair. The first approach is to simply repair each failed node individually until all the failed nodes are repaired successfully. The second approach is to give priority to repairing the stripes that comprise more failed chunks or popularly accessed chunks, so as to meet the requirements on system reliability and access performance.

Heterogeneous environments: RepairBoost can also be adapted to heterogeneous environments. When given the available link bandwidth, RepairBoost can deduce the time to upload and download a chunk at first. It can amend the repair traffic balancing (§3.2) by mapping the vertices to the nodes based on the times for uploading and downloading a chunk (rather than the numbers of uploaded and downloaded chunks in the design for homogeneous environment). This can ensure that the upload and download times are almost the same across the nodes.

RepairBoost then adopts the following ways in the polling mode for transmission scheduling: (i) it pinpoints the node (denoted by N') that spends the least time in uploading and downloading data for repair; (ii) it sorts the set of links \mathcal{L} connected to N' based on the available bandwidth; and (iii) it picks the link from \mathcal{L} that owns the largest upload or download bandwidth for data transmission. RepairBoost repeats the above steps until all the requested chunks are transmitted.

Adaptation to network conditions: RepairBoost can also adapt to network conditions (e.g., random network congestion). Specifically, RepairBoost can break an entire full-node repair process into several subprocesses, each of which repairs a number of single chunks. RepairBoost can then monitor the completion times of nodes in each subprocess to infer the network status, and proactively adjust the repair solutions of next subprocesses. For the node that takes the longest (resp. shortest) time in a subprocess, we can speculatively lessen (resp. increase) the upload and download repair traffic it affords in the next subprocess to balance the repair time across the surviving nodes.

4 Implementation

We implement a prototype of RepairBoost in C++ with around 3,200 lines of codes (LoC), which can serve as an independent middleware running atop existing storage systems to instruct the repair operations on behalf of them. We realize the encoding and decoding functionalities based on the coding library Jerasure v2.0 [2]. We maintain an in-memory key-value store in each node and transmit data through the interfaces of Redis [3].

System architecture: Figure 8 presents the architecture of RepairBoost, which comprises a *coordinator* sitting on the metadata server and multiple *agents* running on the nodes (with one agent per node). The coordinator manages the metadata of stripes (e.g., the mapping from chunks to stripes, and the nodes that the $k + m$ chunks of every stripe reside), while the agents are standby to wait for the repair commands and perform the repair operations cooperatively.

Operating flow: Once a node failure event is reported to the metadata server, the coordinator first pinpoints the IDs of the lost chunks as well as the identities of the associated stripes. It then establishes the *repair solution* of each lost chunk,

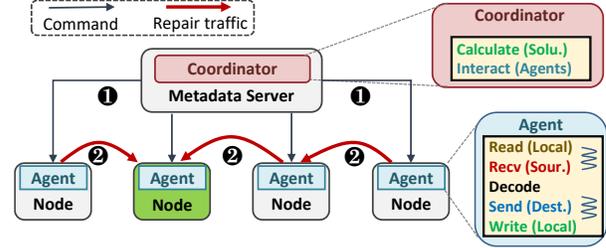


Figure 8: System architecture of RepairBoost. For simplicity, we only illustrate the repair of a single stripe. The node with green color denotes the destination node of this stripe.

including the k surviving chunks selected to participate in the repair, the data routing among the nodes, and the destination node to store the repaired chunk. The repair solutions are sealed into *repair commands* whose format is pre-determined and understood by the agents, which will then be sent to the agents that involve in the repair operations (Step 1).

Upon receiving the repair commands, the agents first extract the repair solutions. They then work cooperatively by (i) reading the requested surviving chunks locally stored, (ii) sending them to the appointed relay nodes, and (iii) decoding (repairing) the lost chunks and storing them locally (Step 2). To accelerate the repair operations, RepairBoost partitions a chunk into many smaller *packets* and uses the multi-threading technique to pipeline the disk I/O, network transmission, and computation in unit of packets.

Integration with Hadoop HDFS: We also integrate RepairBoost into Hadoop HDFS 3.1.4 [6] by adding around 270 LoC². We deploy the coordinator in the NameNode and run agents in the DataNodes. Specifically, when the NameNode is aware of the node failure through periodical heartbeats issued by the DataNodes, the coordinator intercepts the repair commands and calculates the repair solutions via extracting the metadata information (about the addressing of stripes and chunks) from the NameNode³. We then trigger RepairBoost to repair the lost data on behalf of HDFS. To facilitate the integration, RepairBoost also adds a `RepairBoostReconstructor` class in the `erasurecode` package of the DataNode, which collects the repaired chunks from its agent and writes them to the underlying storage of HDFS.

5 Performance Evaluation

We carry out extensive testbed experiments to evaluate the performance of RepairBoost. We summarize our major findings as follows: (i) RepairBoost can improve the repair throughput by 35.0-97.1% (§5.2-§5.4); (ii) RepairBoost can assist the

²Although Hadoop HDFS 3.1.4 employs Intel ISA-L [1] for erasure coding realizations, we can still employ Jerasure Library to repair the lost data with the same Cauchy matrix [2]. We have confirmed the decoding correctness in our evaluation.

³RepairBoost accesses the metadata of the Namenode via executing the command `"hdfs fsck / -files -blocks -locations"`.

repair for a variety of erasure codes and repair algorithms (§5.3); (iii) RepairBoost is more advantageous in the environment with lower network bandwidth (§5.2); (iv) RepairBoost retains its effectiveness when being used in heterogeneous environments and multi-failure repair (§5.4).

5.1 Setup

Testbed and preparations: We evaluate the performance of RepairBoost on Amazon EC2 [8] to unveil its performance in a real-world cloud scenario. We set up 17 virtual machine instances with the type of `m5.large` in the US East (North Virginia) region. Each instance runs Ubuntu 16.04.7 LTS, and is equipped with two vCPUs with 2.5 GHz Intel Xeon Platinum, 8 GB RAM, and 40 GB of EBS storage. The network bandwidth between any two instances is around 1 Gb/s (measured by `iperf`) and the disk bandwidth is around 130 MB/s. Among the 17 instances, we run the RepairBoost coordinator on one instance and deploy the RepairBoost agents on the remaining 16 instances.

Before triggering the repair, we first warm up the system by writing sufficient data encoded by the selected erasure coding scheme. We then erase the data of one instance to mimic a node failure and launch RepairBoost for data repair. We measure the latency, from the time when the failure event is reported to the time when all the lost data is repaired and persisted. We mainly focus on the *repair throughput*, defined as the size of data repaired per time unit. A higher repair throughput indicates a shorter window of vulnerability and hence stronger data reliability.

Erasure codes and repair algorithms: We demonstrate the flexibility, generality, and effectiveness of RepairBoost via deploying RepairBoost atop the following representative erasure codes and repair algorithms. We mainly consider three representative erasure codes: (i) the conventional RS codes [51] that are popularly used in today’s storage systems; (ii) LRCs [21, 52] that trade additional storage overhead for reducing the repair traffic, and (iii) Butterfly code [44], a systematic *minimum storage regenerating* (MSR) code with the minimum repair traffic for a single-node repair.

We also focus on three typical repair algorithms: (i) the conventional repair that transmits k chunks directly to the destination node for repair; (ii) PPR [40], which decomposes a single-chunk repair into sub-stages and exploits the execution parallelism of the sub-stages; and (iii) ECPipe [32], which partitions a chunk into equal-sized slices and pipelines their transmission across the surviving nodes for repair.

Selection of baseline: To select an appropriate baseline approach for comparison, we consider two candidates: a *random selection* (RAN) approach and an LRU-based selection approach. Specifically, the random selection randomly chooses k nodes for data retrieval from the $k + m - 1$ nodes that store the surviving chunks of the corresponding stripe, and also a destination node for data repair from the remaining nodes (§3.2). In the LRU-based selection, we track the timestamp of

Code	CR		ECPipe		PPR	
	RAN	LRU	RAN	LRU	RAN	LRU
RS(6,3)	1.64	1.65	1.25	1.44	1.30	1.43
RS(10,4)	1.67	1.82	1.14	1.41	1.25	2.01
LRC(6,2,2)	1.67	1.68	1.34	1.32	1.46	1.37
LRC(8,2,2)	1.63	1.65	1.25	1.26	1.33	1.43
RC(4,2,5)	1.71	1.74	1.29	1.31	1.37	1.32
RC(5,3,6)	1.70	1.73	1.25	1.30	1.33	1.35

Table 1: Comparison of the random selection and LRU-based selection in terms of their load balancing degrees.

each node when it was last selected for repair, and choose the k nodes for data retrieval from the $k + m - 1$ corresponding nodes and a destination one for data repair with the smallest timestamps. When the nodes to be selected have the same timestamp, the LRU-based selection always chooses the one with the smallest node ID.

We evaluate the *load balancing degrees* of the two approaches as $\frac{L_{max}}{L_{balanced}}$, where L_{max} represents the maximum upload and download repair traffic of a node across the system, while $L_{balanced}$ denotes the upload (or download) repair traffic that is balanced on a node. We can determine $L_{balanced}$ by dividing the amount of repair traffic by the number of surviving nodes.

We consider two representative configurations for each erasure code: RS(6,3) (also used in QFS [43] and Hadoop HDFS [6]), RS(10,4) (used in Facebook f4 [41]), LRC(6,2,2) (also deployed in Windows Azure Storage [21]), LRC(8,2,2) [28], RC(4,2,5)⁴ [44] and RC(5,3,6) (a high-rate MSR code that requires $d = k + 1$ [53]). We randomly distribute the chunks of a stripe across 16 nodes and repeat the test for five runs.

Table 1 gives the load balancing degrees of both the random selection and the LRU-based selection. We see that the random selection generates more balanced repair traffic in most cases. The reason is that the LRU-based selection prioritizes the nodes that have the smallest timestamps and IDs, and the repair traffic becomes progressively imbalanced when more chunks are repaired. Thus, we choose the random selection as the baseline for comparison.

Default configurations: Unless otherwise specified, we select the following default configurations throughout the evaluation. We set the chunk size to 64 MB (the default value in Hadoop HDFS) and the packet size to 1 MB. We mainly consider the following erasure codes: RS(6, 3) [6, 43], LRC(6, 2, 2) [21], and Butterfly(4, 2) [44]. We repair 100 chunks in each test and repeat it for five runs.

5.2 Experiments on Sensitivity

We first study the effectiveness of the parameters configured in RepairBoost on the actual repair performance.

⁴We use $RC(k, m, d)$ to represent the regenerating code that encodes k data chunks into $k + m$ coded chunks, where a data chunk can be repaired by retrieving data from any $d \geq k$ surviving coded chunks [13].

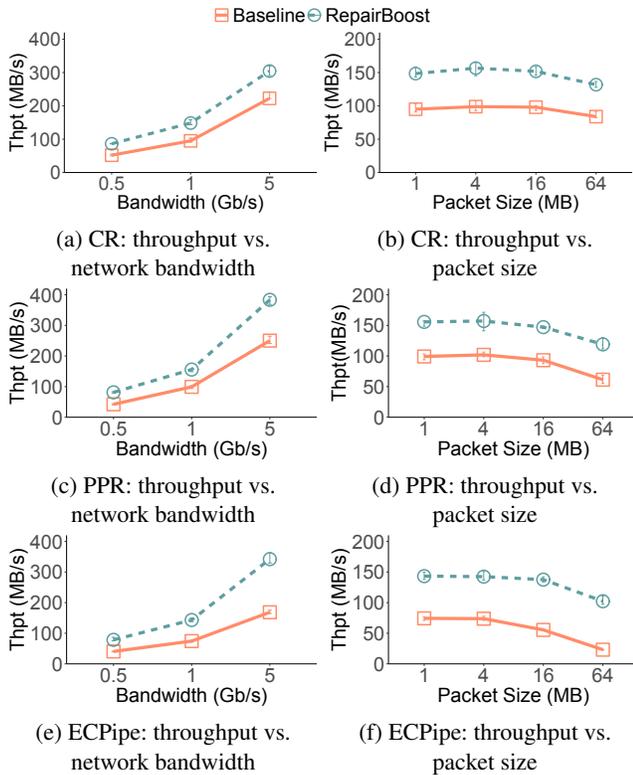


Figure 9: Experiment 1 (Impact of network bandwidth and packet size).

Experiment 1 (Impact of network bandwidth and packet size): We first assess the impact of network bandwidth on the repair throughput when coupling RepairBoost with existing repair algorithms. We vary the network bandwidth from 0.5 Gb/s (i.e., the network bandwidth dominates the repair) to 5 Gb/s (i.e., the disk bandwidth dominates the repair), and evaluate the repair throughput in different repair scenarios.

Figures 9(a), 9(c), and 9(e) show that the repair throughput generally increases with the available network bandwidth, as more data can be transmitted per time unit. Overall, RepairBoost can improve the repair throughput by 72.3% on average for different repair algorithms when compared to the baseline. In addition, we identify that RepairBoost is more advantageous in the network-bandwidth-dominated scenario, as RepairBoost balances the repair traffic and arranges the data transmission for maximizing the utilization of network bandwidth in repair. Specifically, the improvement of RepairBoost on the repair throughput increases from 53.0% (when the network bandwidth is 5 Gb/s) to 96.4% (when the network bandwidth is 0.5 Gb/s). It is worth noting that RepairBoost mainly considers the application scenario where the data transfers across the network dominate the repair performance, which conforms to the assumptions made by many previous studies [19, 40, 56, 58]. Even in the scenario with high-performance network systems (e.g., InfiniBand [45]) for fast memories (e.g., Intel Optane DIMM [61]), RepairBoost can still maintain its performance gain, since the network

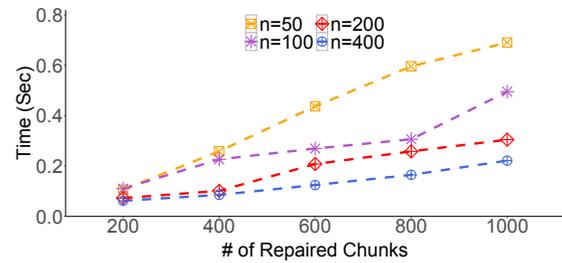


Figure 10: Experiment 2 (Computation time).

bandwidth is likely to be more stringent than the aggregated memory bandwidth (e.g., 44.8 Gb/s of the network bandwidth in RDMA versus 84.8 Gb/s of the write bandwidth in six Optane DIMMs [26]).

As RepairBoost employs multi-threading to pipeline the disk I/O and data transmission in repair, we also study the impact of packet size on the repair performance, where the packet size is changed from 1 MB (i.e., 64 packets per chunk) to 64 MB (i.e., one packet per chunk).

Figures 9(b), 9(d), and 9(f) indicate that the repair throughput decreases when the packet size increases, since the available disk and network bandwidth can be more extensively utilized in repair when a chunk comprises more packets. The multi-threading feature has no effect when the packet size reaches 64 MB (i.e., the chunk size), thereby leading to the lowest repair throughput. Overall, RepairBoost improves the repair throughput by 97.1% compared to the baseline under different packet sizes.

Experiment 2 (Computation time): We allocate one instance on Amazon EC2 with the same configurations as described in §5.1. We measure the computation time needed by RepairBoost to generate the repair solutions under different numbers of nodes (denoted by n) and repaired chunks.

Figure 10 shows the results. We make three observations. First, when the number of nodes is fixed, the computation time of RepairBoost gradually increases with the number of chunks being repaired, as RepairBoost has to process more RDAGs in the repair traffic balancing and transmission scheduling. Second, when the number of repaired chunks is fixed, the computation time drops when more nodes can participate in the repair, as RepairBoost can dispatch more chunks at a time. Third, the computation time needed by RepairBoost is always less than 0.9 seconds, thereby demonstrating that it is qualified to be deployed in the online repair scenario.

As the coordinator is only in charge of solving for the repair solutions and interacting with agents (§4), this experiment can evaluate the scalability of RepairBoost to some extent. When RepairBoost is deployed in a large-scale system (e.g., with thousands of nodes), we offer two options to further reduce the computation time. First, we can break a full-node repair into several subprocesses and iteratively repair a small number of chunks for each subprocess. Second, we can compute the repair solutions in advance and perform the repair based on the pre-computed results when a failure happens.

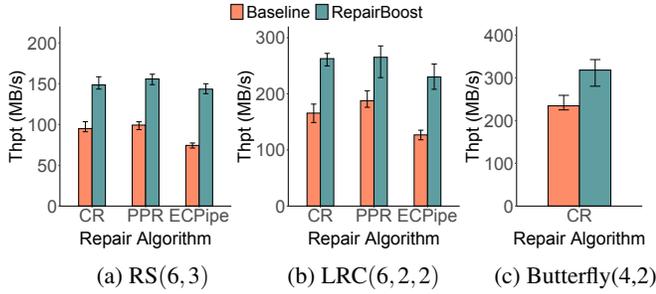


Figure 11: Experiment 3 (Generality).

5.3 Experiments on RepairBoost Property

We also analyze the properties of RepairBoost, in terms of generality and technique breakdown.

Experiment 3 (Generality): We validate the generality of RepairBoost using a variety of representative erasure codes. Given an erasure code, we measure the repair throughput when RepairBoost is coupled with different repair algorithms. Note that we only consider CR for Butterfly(4,2) as it repairs a chunk with multiple sub-chunks coming from different surviving chunks.

Figure 11 shows that all the repair algorithms assisted by RepairBoost can boost the repair for different erasure codes, thereby demonstrating the generality of RepairBoost. Overall, RepairBoost can improve the repair throughput by 60.4% on average for different erasure codes when compared to the baseline. RS(6,3) has the lowest repair throughput (Figure 11(a)), as most of the time it needs to retrieve the most chunks for repair. While LRC(6,2,2) comprises the same number of data chunks (i.e., k) as RS(6,3) within a stripe, it achieves a higher repair throughput (Figure 11(b)), as it only retrieves three chunks to accomplish most of a single-chunk repair. Butterfly(4,2) reaches the highest repair throughput (Figure 11(c)), as it needs to fetch only half of the remaining data (i.e., $\frac{k+1}{2}$ surviving chunks on average per failed chunk).

We also identify that ECPipe reaches the lowest repair throughput for both RS(6,3) and LRC(6,2,2). The root cause is that an RDAG under ECPipe only has one chunk to be transmitted at a time, hence limiting the repair parallelism.

Experiment 4 (Breakdown analysis): We decompose RepairBoost to demonstrate the effectiveness of each designed technique. For simplicity of presentation, we abbreviate the techniques of RepairBoost as follows: (i) *repair traffic balancing* (RTB), which balances the upload and download repair traffic (§3.2) without performing transmission scheduling, and (ii) *transmission scheduling* (TS), which simply schedules data transmission (§3.3) without considering repair traffic balancing.

Figure 12 shows the repair throughput of the baseline, RTB, TS, and RepairBoost. We make two observations. First, the effectiveness of RTB and TS varies across different repair algorithms. For example, TS outperforms RTB for PPR and ECPipe, but reaches a lower repair throughput for CR. Sec-

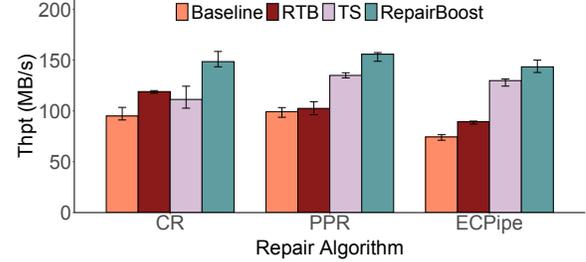


Figure 12: Experiment 4 (Breakdown analysis).

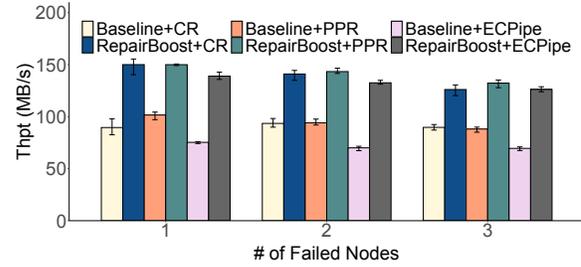


Figure 13: Experiment 5 (Performance on multi-node repair).

ond, RepairBoost always reaches the highest repair throughput, indicating that the two techniques in RepairBoost are complementary without comprising the effectiveness of each other. Specifically, RepairBoost achieves 45.7% and 19.8% higher repair throughput than RTB and TS, respectively.

5.4 Experiments on Practicality

We further assess the practicality of RepairBoost by measuring its performance when tackling multi-failure repair and the single-node failure in heterogeneous environments.

Experiment 5 (Performance on multi-node repair): We extend RepairBoost and study its performance when repairing multiple failed nodes. We erase the data stored in a number of nodes to mimic multiple node failures. We then schedule the RDAGs of the lost chunks for repair. As RS(6,3) can tolerate at most three node failures, this experiment measures the repair throughput when the number of failed nodes increases from one to three.

Figure 13 indicates that RepairBoost also speeds up the repair for multiple failures. Specifically, RepairBoost can improve the repair throughput by 39.5% (when tackling a single node failure) and by 35.7% (when tackling triple node failures). The repair throughput gained by RepairBoost drops slightly when more nodes fail, as fewer surviving nodes can be selected for RepairBoost to balance repair traffic and schedule transmission.

Experiment 6 (Performance with bandwidth heterogeneity): We finally assess the performance of RepairBoost in the presence of bandwidth heterogeneity. We organize the 16 instances with agents into four clusters, where each cluster comprises four instances and the intra-cluster bandwidth is 1 Gb/s. We then use the Linux bandwidth control tool `tc` to throttle the inter-cluster bandwidth for producing the bandwidth diver-

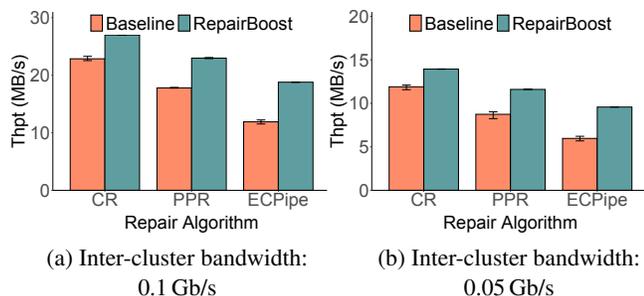


Figure 14: Experiment 6 (Performance with bandwidth heterogeneity).

sity phenomenon in data centers [10, 15]. Specifically, we vary the inter-cluster bandwidth from 0.05 Gb/s to 0.1 Gb/s, such that the over-subscription ratio (i.e., calculated by dividing the intra-cluster bandwidth by the inter-cluster bandwidth) ranges from 10 to 20, as shown in prior studies [10, 15, 55].

Figure 14 shows that RepairBoost retains its effectiveness on accelerating the full-node repair in heterogeneous environments, especially when the inter-cluster bandwidth is more scarce. Specifically, the improvement of repair throughput gained by RepairBoost is 35.0% when the inter-cluster bandwidth is 0.1 Gb/s (Figure 14(a)), and increases to 36.8% when the inter-cluster bandwidth reduces to 0.05 Gb/s (Figure 14(b)).

6 Related Work

We review existing studies to accelerate full-node repair from the following aspects: repair-efficient codes, repair algorithms, and proactive repair with erasure coding.

Repair-efficient codes: LRCs [21, 52] and Rotated-RS codes [27] reduce the repair traffic via loosening the storage efficiency requirement (i.e., increasing a handful of storage overhead). Regenerating codes [13] alleviate the repair traffic by requiring the surviving nodes to send a linear combination of the locally stored data or requiring more surviving nodes to participate in the repair (e.g., product-matrix MSR codes [49]). Butterfly code [44] and Clay code [58] can directly send the locally stored data for repair. Hitchhiker [48] makes the chunks across stripes dependent in the code construction for reducing the repair traffic. Different from the concrete constructions of erasure codes, RepairBoost is a scheduling approach that can assist the full-node repair for a variety of erasure codes.

Repair algorithms: Degraded-first scheduling [31] proposes to launch the degraded read tasks with a higher priority to leverage the unused network bandwidth. PUSH [22] pipelines the transmission of the requested chunks to alleviate the network congestion in repair. PPR [40] partitions an entire repair solution into small sub-stages and executes them in parallel. ECPipe [32] further partitions a chunk into smaller slices and pipelines the transmission of the slices, such that the complexity of the repair time approaches to $O(1)$ in homogeneous

environment. In view of the bandwidth diversity in hierarchical data centers, CAR [56] and ClusterSR [55] reduce and balance the inter-cluster repair traffic, which is considered more scarce than intra-cluster repair traffic. ECWide [18] addresses the repair of wide stripes with ultra-low redundancy in hierarchical data centers. Most of these studies pay special attention to the single-chunk repair, while RepairBoost can help different repair algorithms accelerate the full-node repair.

Proactive repair with erasure coding: Existing mature machine-learning-based failure prediction often take input the SMART attributes [11, 17, 30, 38, 64] combined with additional system events [60] and performance metrics [37], and exhibit high prediction accuracy (e.g., at least 95% [11, 30, 39, 64]) with low false alarm rate (also called false positive rate, e.g., up to 2.5% [38], 0.2-0.4% [63]). FastPR [54] couples migration and erasure-coding-based repair to accelerate the proactive repair. Hu et al. [20] suggest proactively launching degraded reads to bypassing the hotspots, so as to reduce the tail latency in read operations. As an orthogonal study, RepairBoost is also applicable to the proactive repair that employs erasure coding for data recovery.

7 Conclusion

Erasure coding is a storage-efficient means to assure data reliability, yet it is prone to magnify the repair traffic. We present RepairBoost, a scheduling framework that boosts the full-node repair for various erasure codes and repair algorithms. RepairBoost employs a graph abstraction, called an RDAG, to characterize the single-chunk repair solution. It then carefully assigns the repair tasks of the RDAGs to the nodes for balancing the upload and download repair traffic. RepairBoost further schedules the transmission of chunks to saturate the unoccupied bandwidths. Extensive experiments on Amazon EC2 demonstrate the generality, flexibility, and effectiveness of RepairBoost.

Acknowledgments

We thank our shepherd, Mike Mesnier, and the anonymous reviewers for the comments on our camera-ready preparations. This work is supported by Natural Science Foundation of China (62072381, 61832011), CCF-Tencent Open Fund WeBank Special Fund, Xiamen Youth Innovation Fund (3502Z20206052), Zhejiang Lab (2021KF0AB01), the Natural Science Foundation of Fujian Province of China (2020J01002), and Research Grants Council of Hong Kong (AoE/P-404/18).

References

- [1] Intel(R) Intelligent Storage Acceleration Library. <https://github.com/intel/isa-1>.

- [2] Jerasure: Erasure Coding Library. <https://jerasure.org/>.
- [3] Redis. <https://redis.io/>.
- [4] Erasure Coding in Ceph. <https://ceph.com/planet/erasure-coding-in-ceph/>, 2014.
- [5] Yahoo Cloud Object Store - Object Storage at Exabyte Scale. <https://yahooeng.tumblr.com/post/116391291701/yahoo-cloud-object-store-object-storage-at-2015>.
- [6] Apache Hadoop 3.1.4. <https://hadoop.apache.org/docs/r3.1.4/>, 2020.
- [7] HDFS Erasure Coding. <https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSERasureCoding.html>, 2020.
- [8] Amazon EC2. <https://aws.amazon.com/ec2/>, 2021.
- [9] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters. In *Proc. of ACM EuroSys*, 2011.
- [10] T. Benson, A. Akella, and D. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. of ACM IMC*, 2010.
- [11] M. M. Botezatu, I. Giurgiu, J. Bogojeska, and D. Wiesmann. Predicting Disk Replacement towards Reliable Data Centers. In *Proc. of ACM SIGKDD*, 2016.
- [12] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica. HUG: Multi-Resource Fairness for Correlated and Elastic Demands. In *Proc. of USENIX NSDI*, 2016.
- [13] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Transactions on Information Theory*, 56(9):4539–4551, 2010.
- [14] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *Proc. of ACM SOSP*, 2003.
- [15] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proc. of ACM SIGCOMM*, 2009.
- [16] G. Hamerly and C. Elkan. Bayesian Approaches to Failure Prediction for Disk Drives. In *Proc. of ICML*, 2001.
- [17] S. Han, P. P. C. Lee, Z. Shen, C. He, Y. Liu, and T. Huang. Toward Adaptive Disk Failure Prediction via Stream Mining. In *Proc. of IEEE ICDCS*, 2020.
- [18] Y. Hu, L. Cheng, Q. Yao, P. P. Lee, W. Wang, and W. Chen. Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage. In *Proc. of USENIX FAST*, 2021.
- [19] Y. Hu, X. Li, M. Zhang, P. P. C. Lee, X. Zhang, P. Zhou, and D. Feng. Optimal Repair Layering for Erasure-Coded Data Centers: From Theory to Practice. *ACM Transactions on Storage*, 13(4):33, 2017.
- [20] Y. Hu, Y. Wang, B. Liu, D. Niu, and C. Huang. Latency Reduction and Load Balancing in Coded Storage Systems. In *Proc. of ACM SoCC*, 2017.
- [21] C. Huang, H. Simitci, Y. Xu, et al. Erasure Coding in Windows Azure Storage. In *Proc. of USENIX ATC*, 2012.
- [22] J. Huang, X. Liang, X. Qin, Q. Cao, and C. Xie. Push: A Pipelined Reconstruction I/O for Erasure-Coded Storage Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 26(2):516–526, 2014.
- [23] Q. Huang, K. Birman, R. Van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An Analysis of Facebook Photo Caching. In *Proc. of ACM SOSP*, 2013.
- [24] A. Itai, Y. Perl, and Y. Shiloach. The Complexity of Finding Maximum Disjoint Paths with Length Constraints. *Networks*, 12(3):277–286, 1982.
- [25] S. Kadekodi, K. Rashmi, and G. R. Ganger. Cluster Storage Systems Gotta Have HeART: Improving Storage Efficiency by Exploiting Disk-Reliability Heterogeneity. In *Proc. of USENIX FAST*, 2019.
- [26] A. Kalia, D. Andersen, and M. Kaminsky. Challenges and Solutions for Fast Remote Persistent Memory Access. In *Proc. of ACM SoCC*, 2020.
- [27] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang. Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads. In *Proc. of USENIX FAST*, 2012.
- [28] O. Kolosov, G. Yadgar, M. Liram, I. Tamo, and A. Barg. On Fault Tolerance, Locality, and Optimality in Locally Repairable Codes. In *Proc. of USENIX ATC*, 2018.
- [29] H. Li, Y. Zhang, D. Li, Z. Zhang, S. Liu, P. Huang, Z. Qin, K. Chen, and Y. Xiong. URSA: Hybrid Block Storage for Cloud-Scale Virtual Disks. In *Proc. of EuroSys*, 2019.
- [30] J. Li, X. Ji, Y. Jia, B. Zhu, G. Wang, Z. Li, and X. Liu. Hard Drive Failure Prediction Using Classification and Regression Trees. In *Proc. of IEEE/IFIP DSN*, 2014.
- [31] R. Li, P. P. Lee, and Y. Hu. Degraded-First Scheduling for Mapreduce in Erasure-Coded Storage Clusters. In *Proc. of IEEE/IFIP DSN*, 2014.
- [32] R. Li, X. Li, P. P. C. Lee, and Q. Huang. Repair Pipelining for Erasure-Coded Storage. In *Proc. of USENIX ATC*, 2017.
- [33] X. Li, R. Li, P. P. C. Lee, and Y. Hu. OpenEC: Toward Unified and Configurable Erasure Coding Management

- in Distributed Storage Systems. In *Proc. of USENIX FAST*, 2019.
- [34] A. Liaw, M. Wiener, et al. Classification and Regression by randomForest. *R news*, 2(3):18–22, 2002.
- [35] Q. Lin, K. Hsieh, Y. Dang, H. Zhang, K. Sui, Y. Xu, J.-G. Lou, C. Li, Y. Wu, R. Yao, et al. Predicting Node Failure in Cloud Service Systems. In *Proc. of ESEC/FSE*, 2018.
- [36] H. Liu, M. Mukerjee, C. Li, et al. Scheduling Techniques for Hybrid Circuit/Packet Networks. In *Proc. of ACM CoNEXT*, 2015.
- [37] S. Lu, B. Luo, T. Patel, Y. Yao, D. Tiwari, and W. Shi. Making Disk Failure Predictions SMARTer! In *Proc. of USENIX FAST*, 2020.
- [38] A. Ma, F. Douglass, G. Lu, D. Sawyer, S. Chandra, and W. Hsu. RAIDShield: Characterizing, Monitoring, and Proactively Protecting Against Disk Failures. In *Proc. of USENIX FAST*, 2015.
- [39] F. Mahdisoltani, I. Stefanovici, and B. Schroeder. Proactive Error Prediction to Improve Storage System Reliability. In *Proc. of USENIX ATC*, 2017.
- [40] S. Mitra, R. Panta, M. Ra, and S. Bagchi. Partial-Parallel-Repair (PPR): A Distributed Technique for Repairing Erasure Coded Storage. In *Proc. of ACM EuroSys*, 2016.
- [41] S. Muralidhar, W. Lloyd, S. Roy, et al. f4: Facebook’s Warm Blob Storage System. In *Proc. of USENIX OSDI*, 2014.
- [42] J. F. Murray, G. F. Hughes, and K. Kreutz-Delgado. Machine Learning Methods for Predicting Failures in Hard Drives: A Multiple-Instance Application. *Journal of Machine Learning Research*, 2005.
- [43] M. Ovsianikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. *Proceedings of the VLDB Endowment*, 6(11):1092–1101, 2013.
- [44] L. Pamies-Juarez, F. Blagojevic, R. Mateescu, C. Gyuot, E. E. Gad, and Z. Bandic. Opening the Chrysalis: On the Real Repair Performance of MSR Codes. In *Proc. of USENIX FAST*, 2016.
- [45] G. F. Pfister. An Introduction to the Infiniband Architecture. *High performance mass storage and parallel I/O*, 42(617-632):102, 2001.
- [46] J. Plank, S. Simmerman, and C. Schuman. Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications-Version 1.2. *University of Tennessee, Tech. Rep. CS-08-627*, 23, 2008.
- [47] J. S. Plank and C. Huang. Tutorial: Erasure Coding for Storage Applications. Slides presented at USENIX FAST 2013, Feb 2013.
- [48] K. Rashmi, N. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A Hitchhiker’s Guide to Fast and Efficient Data Reconstruction in Erasure-Coded Data Centers. In *Proc. of ACM SIGCOMM*, 2015.
- [49] K. V. Rashmi, N. Shah, and P. V. Kumar. Optimal Exact-Regenerating Codes for Distributed Storage at The MSR and MBR Points via A Product-Matrix Construction. *IEEE Transactions on Information Theory*, 57(8):5227–5239, 2011.
- [50] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook-Warehouse Cluster. In *Proc. of USENIX HotStorage*, 2013.
- [51] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.
- [52] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring Elephants: Novel Erasure Codes for Big Data. *Proceedings of the VLDB Endowment*, 6(5):325–336, 2013.
- [53] N. B. Shah, K. Rashmi, P. V. Kumar, and K. Ramchandran. Interference Alignment in Regenerating Codes for Distributed Storage: Necessity and Code Constructions. *IEEE Transactions on Information Theory*, 58(4):2134–2158, 2011.
- [54] Z. Shen, X. Li, and P. P. C. Lee. Fast Predictive Repair in Erasure-Coded Storage. In *Proc. of IEEE/IFIP DSN*, 2019.
- [55] Z. Shen, J. Shu, Z. Huang, and Y. Fu. ClusterSR: Cluster-Aware Scattered Repair in Erasure-Coded Storage. In *Proc. of IEEE IPDPS*, 2020.
- [56] Z. Shen, J. Shu, and P. P. C. Lee. Reconsidering Single Failure Recovery in Clustered File Systems. In *Proc. of IEEE/IFIP DSN*, 2016.
- [57] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.
- [58] M. Vajha, V. Ramkumar, B. Puranik, G. Kini, E. Lobo, B. Sasidharan, P. V. Kumar, A. Barg, M. Ye, S. Narayana-murthy, et al. Clay Codes: Moulding MDS Codes to Yield an MSR Code. In *Proc. of USENIX FAST*, 2018.
- [59] H. Weatherspoon and J. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *Proc. of IPTPS*, 2002.
- [60] Y. Xu, K. Sui, R. Yao, H. Zhang, Q. Lin, Y. Dang, P. Li, K. Jiang, W. Zhang, J.-G. Lou, et al. Improving Service Availability of Cloud Systems by Predicting Disk Error. In *Proc. of USENIX ATC*, 2018.
- [61] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An Empirical Guide to the Behavior and

Use of Scalable Persistent Memory. In *Proc. of USENIX FAST*, 2020.

- [62] H. Zhang, M. Dong, and H. Chen. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *Proc. of USENIX FAST*, 2016.
- [63] J. Zhang, P. Huang, K. Zhou, M. Xie, and S. Schelter. HDDse: Enabling High-Dimensional Disk State Embedding for Generic Failure Detection System of Heterogeneous Disks in Large Data Centers. In *Proc. of USENIX ATC*, 2020.
- [64] B. Zhu, G. Wang, X. Liu, D. Hu, S. Lin, and J. Ma. Proactive Drive Failure Prediction for Large Scale Storage Systems. In *Proc. of IEEE MSST*, 2013.

KVIMR: Key-Value Store Aware Data Management Middleware for Interlaced Magnetic Recording Based Hard Disk Drive

Yuhong Liang, Tsun-Yu Yang, and Ming-Chang Yang
The Chinese University of Hong Kong

Abstract

Log-Structured Merge-Tree (LSM-tree) based key-value (KV) store provides write-intensive applications with high throughput on Hard Disk Drive (HDD). Recently, the emerging Interlaced Magnetic Recording (IMR) technology makes the IMR based HDD become another desirable option to construct a cost-effective KV store because of its high areal density. Nevertheless, we observe that deploying LSM-tree based KV store on IMR based HDD may suffer noticeable degradation on throughput of incoming reads/writes. Thus, this paper presents KVIMR, a data management for constructing a cost-effective yet high-throughput LSM-tree based KV store on IMR based HDD. KVIMR is architected as a *middleware*, interposed between LSM-tree based KV store and IMR based HDD, to embrace the compatibility for mainstream LSM-tree based KV store implementations with limited modifications. Technically, KVIMR adopts a novel *Compaction-aware Track Allocation* scheme, which leverages the special properties behind the compaction process to remedy the throughput degradation. KVIMR further utilizes a novel *Merged RMW* approach to improve the efficiency of persisting a multi-track-sized file of KV store into IMR tracks with the ensured crash consistency. Our evaluations on several well-known LSM-tree based KV store implementations reveal that KVIMR not only improves the overall throughput by up to $1.55\times$ under write-intensive workloads but even achieves $2.17\times$ higher throughput under high space usage of HDD, as compared with the state-of-the-art track allocation scheme for IMR.

1 Introduction

Persistent key-value (KV) stores have gained popularity in diverse data-intensive applications, ranging from electronic commerce [15], internet services [43], to cloud environments [13, 33], because of its high efficiency in insertions, point and range queries, and deletions. Among various implementations of KV store, Log-Structured Merge-Tree (LSM-tree) [38] based KV stores (e.g., BigTable [13], Cassandra [34], LevelDB [21], HBase [26], HyperLevelDB [17] and

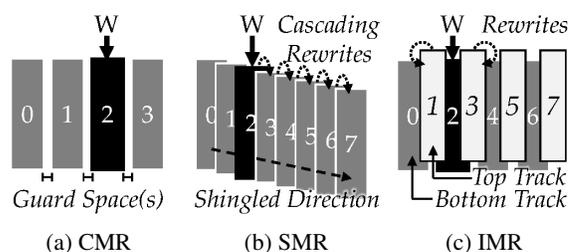


Figure 1: Track Layouts of CMR, SMR, and IMR.

RocksDB [18]) demonstrate its strength in delivering high write throughput on the mechanical hard-disk drive (HDD). The reason is that the LSM-tree takes advantage of the sequential access strength of HDD [38] by first batching the inserted KV pairs in memory and then persisting the batched data from memory into HDD sequentially.

With the explosive growth of data, how to construct a *cost-effective* KV store has become another vital challenge. However, the most common type of HDD, i.e., the *Conventional Magnetic Recording (CMR)* based HDD, has reached its bottleneck in providing higher areal density [11] to lower its *cost-per-GB* because of the super-paramagnetic effect [37]. Among various novel technologies that can break such areal density limitation [12, 16, 28, 30–32, 35, 40, 44, 46, 55], *Shingled Magnetic Recording (SMR)* technology [10, 47] is an eligible alternative of CMR for offering higher areal density gain with limited manufacturing changes [49, 51]. Different from the CMR track layout in which a small *guard space* is introduced to every two physically-adjacent tracks (as shown in Figure 1a), every SMR track is partly overlapped by its subsequent one(s), making that SMR tracks are placed closer to each other (as shown in Figure 1b). However, with this “shingled” track layout, data updates to a single SMR track may incur the time-consuming *track rewrites* over a great amount of subsequent tracks along the shingled direction, so as to avoid losing any valid data [10, 51]. Although lots efforts have been made to tackle the track rewrite issue for SMR based HDD at different layers [10, 27, 39, 41, 42, 51–53], *the applicability of SMR based HDD is still limited because of its long write tail-latency* [51].

More recently, the emerging *Interlaced Magnetic Recording* (IMR) technology [28] has been validated for being effective in delivering *even-higher* areal density with *reduced* track rewrite issue than the SMR [22], making the IMR become a more desirable successor to the CMR. In general, the areal density gain of IMR is mainly attributed to the linear density increasing of tracks with different laser currents [22, 23]. As shown in Figure 1c, IMR embraces a new “interlaced” track layout to organize tracks into *top tracks* and *bottom tracks*, with each bottom track overlapped partially by two adjacent top tracks. With this interlaced track layout, any top tracks can be freely updated *without* incurring any track rewrites; whilst updates to a bottom track would only introduce *at most two* track rewrites on its two adjacent top tracks. Moreover, to conduct track rewrites with ensured crash consistency, the *read-modify-write* (RMW) approach [25] is presented.

Inspired by the *ever-high* areal density and the *reduced* track rewrite issue of IMR technology, this paper for the first time, seeks for the possibility in constructing a *cost-effective yet high-throughput* LSM-tree based KV store on IMR based HDD. Nevertheless, we observe that deploying LSM-tree based KV stores on IMR based HDD may still suffer noticeable degradation on throughput of incoming reads/writes. The reason is that the RMW process of IMR based HDD may amplify the background I/Os introduced by the compaction process of LSM-tree based KV store, and such amplified background I/Os may thereby degrade the efficiency of the compaction process and further slow down the throughput of incoming reads/writes [9, 39, 53]. Moreover, none of the existing designs for managing IMR tracks is able to well remedy such throughput degradation, because the special software behavior, hid behind compaction process of the LSM-tree based KV store, is out of their design considerations.

With the goal of alleviating the noticeable throughput degradation caused by the IMR technology, this paper presents KVIMR, a data management middleware for constructing a *cost-effective yet high-throughput* LSM-tree based KV store on IMR based HDD. In particular, KVIMR is architected as a *middleware*, sitting between LSM-tree based KV store and IMR based HDD, in order to facilitate 1) the support for various existing implementations of LSM-tree based KV store with limited modifications and 2) the direct and efficient management on IMR based HDD.

To make KVIMR be aware of the software behaviour of LSM-tree based KV store with limited overhead, the most key semantic information (i.e., the “level” information) regarding the data of LSM-tree is passed to KVIMR along with data writes. Given the “level” information as a clue, KVIMR introduces a novel *Compaction-aware Track Allocation* scheme to allocate tracks for data of LSM-tree KV store according to the “level” information. This scheme effectively remedies the throughput degradation and improves the compaction efficiency by 1) minimizing the time-consuming RMWs when persisting data files (i.e., *SSTables*) of LSM-tree based KV

store and 2) efficiently accessing the data files of LSM-tree based KV store during the compaction process. To further improve the compaction efficiency when the time-consuming RMWs are inevitable, KVIMR employs a novel *Merged RMW* approach to efficiently persist a multi-track-sized data file of LSM-tree based KV store into IMR tracks. Its key idea is to re-order the multiple “track-by-track” RMWs into a single “merged” RMW while still nicely ensure the crash consistency. With this approach, KVIMR significantly reduces the number of required `sync`-like functions, which have adverse effects on I/O performance of HDD [2, 24, 45], and avoids redundant track rewrites caused by the track-by-track RMW approach.

KVIMR is developed in C++ and provides a POSIX complaint interface for the existing LSM-tree KV store implementations to interact with IMR based HDD. Specifically, we modify three well-known LSM-tree KV stores, i.e., RocksDB [18], LevelDB [21], and HyperLevelDB [17], by replacing the native file operations with a similar set of file operations provided by KVIMR to access the data files in an emulated IMR based HDD. Our evaluations on these three LSM-tree based KV store implementations reveal that KVIMR not only improves the overall throughput by up to $1.55\times$ under write-intensive workloads but even achieves $2.17\times$ higher throughput under high space usage of HDD, as compared with the state-of-the-art track allocation scheme for IMR.

The rest of this paper is organized as follows: Section 2 presents the background and motivation regarding this work. Section 3 introduces the design of KVIMR. Then, Section 4 provides the implementation details and demonstrates the evaluation results. Finally, Section 5 presents relevant studies and Section 6 concludes this work.

2 Background and Motivation

2.1 LSM-Tree based KV Store

Because of the strength in delivering high write throughput on the mechanical HDD, Log-Structured Merge-Tree (LSM-tree) [38] inspires many well-known key-value (KV) stores, such as RocksDB [18], LevelDB [21], and HyperLevelDB [17]. In general, these LSM-tree based KV stores embrace a similar design concept, borrowed from LSM-tree, on managing KV pairs in the HDD, and support a similar set of KV operations such as `put`, `get` and `delete` operations.

The `put` operation is for inserting KV pairs into the KV store. To deliver high throughput of `put` operations on HDD, LSM-tree based KV store first batches all the inserted KV pairs in an in-memory sorted skiplist namely *Memtable*. When the Memtable exceeds its size limit (e.g., 64 MB in RocksDB [18]), LSM-tree based KV store creates a new Memtable to keep accommodating new KV pairs, while the old Memtable is converted to an immutable in-memory sorted skiplist namely *Immutable Memtable*. In the background, an important thread takes over the Immutable Memtable

and persists it into HDD as an in-disk sorted string table (SSTable). In addition, all the SSTables in the disk are organized into multiples levels (i.e., $L_0 \sim L_n$), similar to the multiple in-disk components of the LSM-tree [38], and the size limit of each level increases exponentially by a factor (e.g., 10 in both LevelDB [21] and RocksDB [18]) from L_1 to larger level(s). Moreover, the SSTable converted from the Immutable Memtable is usually placed at L_0 first; then, once the total size of SSTables of any level L_i exceeds its size limit, the background thread keeps performing the *compaction process* in a cascading way to compact the KV pairs from smaller levels to larger levels, until all levels are within their size limits. Specifically, the compaction process 1) picks one SSTable in L_i , 2) merges it with all SSTables with overlapped key ranges in L_{i+1} , 3) creates new SSTable(s) into L_{i+1} , and finally 4) deletes all stale SSTables from the KV store sooner or later.

Besides of the `put` operation, LSM-tree based KV store also supports the `get` operation to read out the value associated with the given key. Specifically, to find out the latest version of value, LSM-tree based KV store searches the requested key-value pair(s) from Memtable, Immutable Memtable, and SSTables from smaller levels to larger levels in order. Moreover, LSM-tree based KV store also supports the `delete` operation to remove specific KV pair(s) from the KV store.

2.2 Interlaced Magnetic Recording

Interlaced Magnetic Recording (IMR) [28] is a new disk technology that adopts the “interlaced” track layout to increase the areal density by shortening the distance between adjacent tracks. As shown in Figure 1c, tracks of IMR based HDD are organized into *top tracks* and *bottom tracks*, and each bottom track (e.g., Track #2) is overlapped partially by two adjacent top tracks (e.g., Tracks #1 and #3). However, with this interlaced track layout, *writing data into any bottom track will destroy the (valid) data stored in the two adjacent top tracks*. Thus, in order to protect the (valid) data of top tracks against data loss on writing data into bottom tracks, the *read-modify-write (RMW)* approach [25] is introduced; specifically, it ensures the crash consistency by quarantining that the following sequence of “read-modify-write” can be enforced: 1) “Read”: Backing up the valid data of the two adjacent top tracks in a *backup region*; 2) “Modify”: Writing the updated data properly to the bottom track; and 3) “Write”: Re-writing the backed-up valid data back to the adjacent top tracks.

Since the RMW is time-consuming [48], most existing studies for IMR based HDD focus on how to minimize the probability of incurring the RMWs. Specifically, some propose to reduce RMWs via *track allocation* (i.e., how IMR tracks are allocated to accommodate the written data). For example, Gao *et al.* present a *three phase track allocation* to allocate tracks based on three phases of disk space usage [19, 20]: In the first phase, data are placed into *bottom tracks only* until all the bottom tracks are full of data (i.e., 0% ~ 50% space usage); in

the second phase, data are placed into *every other top tracks* until half of the top tracks are used (i.e., 50% ~ 75% space usage); and in the third phase, data are placed into *the rest of top tracks* (i.e., 75% ~ 100% space usage). As a result, the first phase would not incur any RMW(s), while the second phase (resp. to the third phase) ensures at most one (resp. to two) top track(s) will be re-written on writing data into any bottom track. Based on the three phase track allocation, Wu *et al.* further propose a *zigzag track allocation* to reverse the track allocation order in the second phase, making every other top tracks be allocated from inner tracks to outer tracks, for better preserving data locality [48, 50].

Another series of studies tries to minimize the probability of incurring the RMWs by considering the *update frequencies of data*. Notably, in the literature, the *frequently-updated data* (resp. to *less-frequently-updated data*) are also referred to as the *hot data* (resp. to *cold data*). For example, Wu *et al.* propose a *top buffer* design to exploit a few top tracks as “write buffer” to place the hot data, and a *block swap* method to exchange the hot data in bottom tracks with the cold data in top tracks, so as to reduce the RMWs caused by updating hot data in bottom tracks [48, 50]. More recently, Hajkazemi *et al.* propose three new track-level techniques, namely *track flipping*, *selective track caching*, and *dynamic track*, to reduce the amount of writes to bottom tracks by moving hot data to top tracks and cold data to bottom ones in different ways [25]. However, please note that, although time-consuming RMWs can be indeed reduced by considering the update frequencies of data, *not all sorts of applications can be benefited from this series of studies*. For example, the data of LSM-tree based KV stores (i.e., SSTable), which is our target application, are written once but *never updated* before being deleted.

2.3 Motivation: Degradation on Throughput and Compaction Efficiency

In order to investigate how serious the performance of LSM-tree based KV store would be degraded by the time-consuming RMW process of IMR technology, we deploy the widely used RocksDB on an 100 GB, emulated IMR based HDD. In particular, in order to reasonably allocate IMR tracks for accommodating SSTables, we implement the classical *sequential allocation* scheme [41] (denoted as *Seq*), and the state-of-the-art, IMR based *three-phase allocation* scheme [19, 20, 50] (denoted as *3Phase*). Moreover, to fairly compare and demonstrate that how the performance of IMR based *three-phase allocation* is affected by the RMW process, we also deploy RocksDB on an 100 GB CMR based HDD and adopts *three-phase allocation* scheme to allocate CMR tracks for accommodating SSTables (denoted as *CMR*). More details about the implementations can be found in Section 4.1.

Figure 2a shows the performance of randomly loading/inserting 75 millions of 1 KB KV pairs, generated by YCSB [14], into RocksDB, where the x-axis denotes different

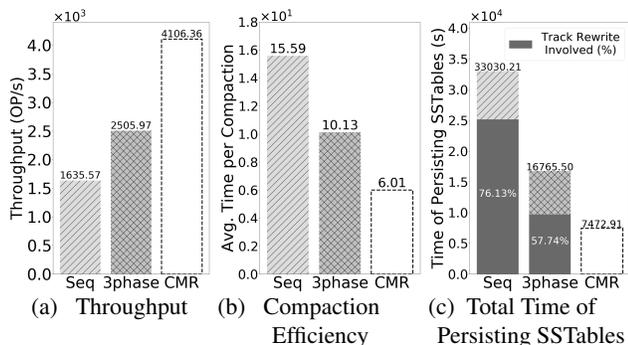


Figure 2: Performance of Loading 75 Millions of KV Pairs into RocksDB under Different Track Allocation Schemes.

schemes of track allocation and the y-axis shows the throughput (i.e., the number of I/Os per second). It can be observed that, although the IMR based 3phase indeed effectively achieves $1.53 \times$ higher throughput than the classical Seq, there still exists a large room for improvement since the IMR based 3phase also suffers 38.97% degradation on throughput as compared to that of CMR. Since it is known that the write throughput of a LSM based KV store could be significantly affected by the compaction process [9, 39, 53], Figure 2b further shows the *compaction efficiency*, which is regarded as the average time required by a compaction process, achieved by different track allocation schemes. It can be observed that, as compared with CMR, Seq and 3phase require $2.59 \times$ and $1.68 \times$ longer time to complete a compaction process on average, and result in the observed degradation on throughput.

Furthermore, based on our investigation, the observed compaction efficiency degradation of Seq and 3phase can be mainly attributed to the time-consuming RMW process of IMR technology. The reason is that the compaction process involves persisting SSTables into HDD, and the time-consuming RMW process could be thereby frequently triggered during the compaction process. Thus, to explicitly demonstrate how the compaction efficiency is affected by the RMW process, Figure 2c shows the total time of persisting SSTables (during the compaction processes) required by different track allocation schemes and indicates the portion of time spent on rewriting top tracks during the RMW process. It can be observed that, when compared with CMR, the total time of persisting SSTables required by Seq and 3phase is noticeably lengthen by around $4.42 \times$ and $2.24 \times$ respectively. The reason is that Seq and 3phase entail 25145.76 and 9681.21 seconds on rewriting top tracks during the RMW process, which account for 76.13% and 57.74% of their respective total time of persisting SSTables. Motivated by such observed degradation on throughput and compaction efficiency, a novel middleware KVIMR is proposed in Section 3 to deliver high throughput on IMR based HDD with limited changes to the well-known LSM-tree based KV store implementations.

3 KVIMR

3.1 System Architecture

As shown in Figure 3, KVIMR is architected as a *middleware* between LSM-tree based KV store and IMR based HDD, so as to facilitate 1) the support for various LSM-tree based KV store implementations with limited modifications and 2) the direct and efficient management on IMR based HDD.

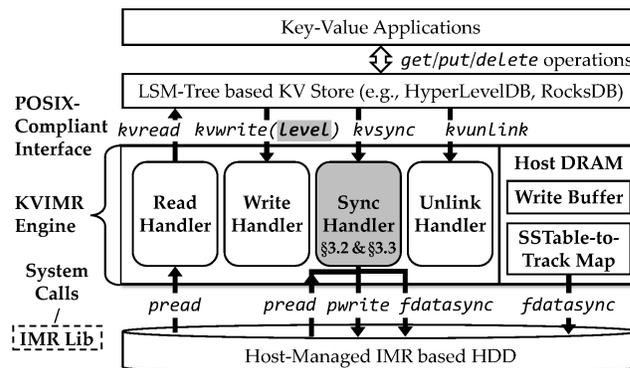


Figure 3: The System Architecture of KVIMR.

To realize this objective, KVIMR provides a POSIX compliant interface [5] so that the upper LSM-tree based KV store can easily access its SSTables through a similar set of common file operations (i.e., kvread/kvwrite/kvsync/kvunlink operations) on IMR based HDD with very limited modifications. Besides, to make the KVIMR be aware of the special *compaction behaviour* behind the LSM-tree based KV store, we promote to pass the “level” information of SSTables along with the kvwrite file operation on writing SSTables. Notably, since the *level* information is one of key design principles of the LSM-tree, it can be widely found in the mainstream LSM-tree based KV store implementations, such as RocksDB [18], LevelDB [21] and HyperLevelDB [17]. Thus, we argue that *the attempt to pass down the level information will not limit the applicability but instead increase the generality and compatibility of KVIMR*. It is also worth noting that, based on our experience in integrating KVIMR with the three aforementioned LSM-tree based KV store implementations, the modifications of replacing POSIX interface (with file operations provided by KVIMR) and passing level information to KVIMR are just about 100 lines of codes for each KV store implementation.

On the other hand, in our implementation, by employing the existing pread/pwrite/fdatasync system calls with the O_DIRECT flag, KVIMR directly manages the data of files (e.g., SSTables) and performs the RMW approach [25] on tracks of the emulated IMR based HDD (please see Section 4.1 for details). Please note that recent researches have demonstrated that the host-managed HDD model and the libzbc interface [4] can simplify and facilitate the direct management on SMR based HDD [52, 53]. We envision such host-managed HDD model and library (referred to as

IMRLib in Figure 3) will also be available for IMR based HDD with the provision of fundamental functions (such as reading/writing/syncing data and exposing track information), so that a unified application programming interface (API) can be offered for the development of KVIMR. We leave the IMRLib extension to KVIMR in the future.

If we take a closer look at Figure 3, the core KVIMR engine maintains an *SSTable-to-Track Map* and allocates a *Write Buffer* in the DRAM space of the host system, and incorporates four *Handlers* to take over the *kvread/kvwrite/kvsync/kvunlink* file operations from the upper LSM-tree based KV store respectively. Their main functionalities and interactions are summarized as follows:

- **SSTable-to-Track Map (or S2TMap)** maintains the one-to-many relationship between an SSTable and multiple IMR tracks allocated for it, because the SSTable size in the mainstream LSM-tree based KV store implementations is usually larger than the IMR track size (which is 2 MB [25, 50]). We implement S2TMap as `map<sstable id, track list>` and only allocate an IMR track for a single SSTable file for simplicity. Notably, since S2TMap is the key metadata of KVIMR, how to maintain and promise the crash consistency of S2TMap will be discussed in detail in Section 3.4.

- **Write Handler** is responsible for temporarily buffering the written SSTable in the *Write Buffer* (in the DRAM space) whenever the background thread (i.e., compaction process) of KV store invokes *kvwrite* operations to pass down the SSTable data into middleware, so that the entire SSTable can be later persisted into the IMR based HDD by the *Sync Handler* in one shot. Notably, at runtime, the *Write Buffer* of KVIMR only holds a few SSTables at any given time, since each compaction process/thread only creates one SSTable at a time and will persist it into the storage by the *Sync Handler*. That is, once an SSTable is persisted, KVIMR immediately releases the corresponding DRAM space to keep low demand of *Write Buffer*.

- **Sync Handler** is in charge of persisting the buffered SSTable into IMR tracks (based on different track allocation schemes) whenever the background thread (i.e., compaction process) of KV store needs to ensure the SSTable data are persisted in HDD by invoking *kvsync* operation. In addition, during the process of SSTable persisting, the *Sync Handler* also performs the RMW approach to rewrite tracks (if needed) to ensure the crash consistency of written data, and builds the relationship between the specified SSTable and the allocated tracks in S2TMap to facilitate the subsequent accesses to persisted SSTables. Moreover, at runtime, to efficiently determine whether the RMW approach is needed, KVIMR also maintains a bitmap in the DRAM space to keep track of the allocation status of tracks. However, this bitmap does not have to be persisted in HDD since it can be easily rebuilt by scanning S2TMap.

- **Read Handler** fetches the requested part of an SSTable from the corresponding track(s) of the IMR based HDD by

looking up the S2TMap whenever the KV store needs to read the content of SSTable by invoking *kvread* operation (e.g., upon performing the compaction process or servicing *get* operations).

- **Unlink Handler** is to remove an SSTable from HDD and label the corresponding track(s) as *free tracks* (i.e., tracks containing *no* valid data) by resetting the corresponding entries in S2TMap whenever the background thread (i.e., compaction process) of KV store invokes *kvunlink* operation to delete an SSTable from HDD.

Among these four handlers, the *Sync Handler* plays the most critical role to affect the throughput of incoming reads/writes, since how SSTables are persisted into tracks of IMR based HDD may largely affect the number of incurred time-consuming RMWs and the efficiency of the background compaction process. Thus, in the following sections, *we shall mainly focus on the design of the Sync Handler*.

3.2 Compaction-aware Track Allocation

Given the *level* information of SSTables as a clue, this section explores how to leverage the special behavior behind compaction process to properly allocate the *two-tier* IMR tracks for accommodating the *multi-level* SSTables. Specifically, Section 3.2.1 first introduces two special properties, namely *compaction frequency* and *compaction locality*, extracted from the compaction process. Then, based on the observed properties and the *level* information of SSTables, Section 3.2.2 introduces a novel *Compaction-aware Track Allocation* scheme to alleviate the throughput degradation by 1) minimizing the time-consuming RMWs and 2) efficiently accessing the SSTables during the compaction process.

3.2.1 Special Properties of Compaction Process

Compaction Frequency. In an LSM-tree based KV store, SSTables of different levels usually have different frequencies of being compacted (i.e., *compaction frequency*) by the compaction process. The reason is twofold: First, as introduced in Section 2.1, the compaction process usually takes place in a cascading way, from smaller levels to larger levels; second, the size limits of different levels increase exponentially along with the levels. As a result, the SSTables of *smaller* levels are more likely to be compacted frequently than that of *larger* levels [29, 54]. In other words, *the lifespan of SSTables of larger levels tends to be longer than that of smaller levels*.

Compaction Locality. Based on our investigation, there exists a very special access locality, namely *compaction locality*, among *different rounds* of compaction process in LSM-tree based KV store. Specifically, as introduced in Section 2.1, a single round of compaction process merges the victim SSTable and the existing SSTables with overlapped key ranges into new SSTables with *close* key ranges. Moreover, since these newly generated SSTables are composed of KV pairs of

close key ranges, they (or part of them) are more likely to be involved (specifically, `read` and `unlink`) in the latter round(s) of compaction process as compared to other SSTables with far key ranges. That is, there exists a special access tendency that the SSTables generated by a round of compaction process are likely to be involved in the latter round(s) of compaction process, and we refer it as *compaction locality*.

3.2.2 Design of Compaction-aware Track Allocation

Based on the two observed special properties, this section presents the design of the *Compaction-aware Track Allocation* scheme, which comprises two major steps, namely *Step 1) Level based Bi-Tiering* and *Step 2) Relaxed-Sequential Track Allocation*, in allocating tracks for an SSTable.

Step 1) Level based Bi-Tiering The first step determines which tier of tracks (i.e., either top tier of tracks or bottom tier of tracks) should be used to accommodate an SSTable based on its level information.

Based on the property of *compaction frequency*, the lifespan of SSTables of larger levels tends to be longer than that of smaller levels. Such property inspires us that, if bottom tracks can be allocated to accommodate SSTables with larger levels (rather than smaller levels), the bottom tracks will be occupied by SSTables with longer lifespan and will not be frequently re-allocated by other SSTables with shorter lifespan. That is, the key idea of the first step is to *allocate bottom tracks to accommodate SSTables of larger levels, so that the probability of incurring the RMWs on allocating bottom tracks could be thereby minimized*.

As shown in Figure 4, in our implementation, we propose to allocate the bottom tracks only for SSTables of the “current largest” level (e.g., L_5), while allocate the top tracks for SSTables of the non-largest levels (e.g., $L_0 \sim L_4$). This is because, given that the total capacity of bottom tracks and top tracks in IMR based HDD are roughly the same [19, 20] while the size limits among levels in LSM-tree based KV store increase exponentially by a factor (which is usually larger than 2 [17, 18, 21, 36]), the bottom tracks would be fully allocated by SSTables of the largest level when the LSM-tree grows.

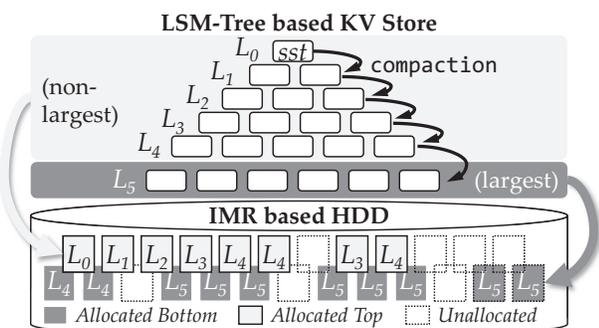


Figure 4: The Level based Bi-Tiering (Step 1).

Moreover, as astute readers may have noticed, when the LSM-tree keeps growing, the size limit of the largest level

may become larger than the total capacity constituted by all bottom tracks; as a result, accommodating all the SSTables of the largest level in bottom tracks may become impossible. However, we argue that *this situation will not contradict the design concept of the first step*. The reason is that, although we have no choice to accommodate some SSTables of the largest level in top tracks, all the bottom tracks can still be allocated only by SSTables of the largest level, so that the probability of incurring the RMWs on allocating bottom tracks can be still effectively minimized.

Step 2) Relaxed-Sequential Track Allocation The second step further determines which tracks in the specific tier (that is decided by the *Level based Bi-Tiering* (i.e., Step 1)) should be allocated to accommodating an SSTable.

The design principle of the second step is inspired by the following two observations. First, according to the property of *compaction locality*, the SSTables generated by a round of compaction process are likely to be involved in the latter round(s) of compaction process. Second, since IMR based HDD adopts the similar rotational disk mechanisms of CMR based HDD, accessing data sequentially is also much efficient than accessing it randomly in IMR based HDD. Thus, we argue that *it may be beneficial to accommodate the SSTables generated by a round of compaction process into tracks “as sequential as possible”*. The reason is that the SSTables generated by a round of compaction process can be sequentially written into tracks with *high sequential write performance*, and also be sequentially compacted by the latter round(s) of compaction process with *high sequential read performance*.

However, in order to comply with the proposed *Level based Bi-Tiering* (i.e., Step 1) and avoid any data migrations or RMWs, a *Relaxed-Sequential Track Allocation* is instead adopted to *relax* the degree of sequentiality on track allocation from two aspects: First, since all the SSTables generated in a round of compaction process must belong to *same* level, the proposed *Level based Bi-Tiering* would suggest accommodating all of these SSTables in the *same* tier of tracks (i.e., either bottom tracks or top tracks). Thus, instead of allocating the tracks in the *strictly-sequential* order (i.e., one bottom track followed by the adjacent top track), we propose to allocate bottom tracks and top tracks *separately* to accommodate SSTables in the *relaxed-sequential* order (i.e., one bottom track followed by the adjacent bottom track, or one top track followed by the adjacent top track). Second, when allocating tracks in the relaxed-sequential order, we propose to further *relax* the degree of sequentiality by “skipping” the track(s) which are currently allocated by other SSTables or may incur the time-consuming RMWs if possible. However, if none of the unallocated/free bottom tracks can be allocated without incurring RMWs, we propose to allocate the nearest unallocated bottom track to best preserve the degree of sequentiality.

3.3 Merged Read-Modify-Write

To further improve the compaction efficiency when the time-consuming RMWs are inevitable, this section explores how to improve the efficiency of persisting the buffered SSTable from the *Write Buffer* into the allocated tracks. Specifically, Section 3.3.1 first introduces the process of persisting the buffered SSTable when the *Naïve RMW* approach [25] is employed. Then, Section 3.3.2 reveals the potential inefficiency of the *Naïve RMW* approach and introduces a novel *Merged RMW* approach which re-orders multiple RMWs into a “merged RMW” with the improved persisting efficiency and the ensured crash consistency.

3.3.1 SSTable Persisting with Naïve RMW

Algorithm 1 shows the process of *SSTable Persisting with Naïve RMW*, which persists the buffered SSTable (denoted as SST) from the *Write Buffer* into the allocated track(s) (denoted as *TrackList*) when the *Naïve RMW* approach [25] is employed. Notably, the notations $SST[i]$ and $TrackList[i]$ denote that the i^{th} track-size content of the SST should be persisted into the i^{th} allocated track in *TrackList*.

First of all, the content of the buffered SSTable are persisted into the allocated tracks on a *track-by-track* basis (i.e., *at the granularity of a track* [25]) (Lines 1~14). Specifically, if track rewrite(s) are incurred when writing $SST[i]$ into $TrackList[i]$, the *Naïve RMW* is performed as follows (Lines 4~12): First, it performs back-up (Lines 5~6), followed by invoking a *sync*-like function to ensure the valid data of the adjacent top track(s) are safely persisted into *BackupRegion* (Line 7); Second, it writes the content of $SST[i]$ into the $TrackList[i]$ (Line 8), followed by invoking a *sync*-like function to ensure $SST[i]$ is safely persisted into the IMR based HDD (Line 9); Thirdly, it performs move-back (Lines 10~11), followed by invoking a *sync*-like function to ensure the backed-up valid data are safely persisted into adjacent top tracks (Line 12). Otherwise, if track rewrite(s) are not incurred when writing $SST[i]$ into $TrackList[i]$, $SST[i]$ are directly written into the $TrackList[i]$ (Lines 13~14). Finally, a *sync*-like function is invoked to ensure that the entire SST is safely persisted into the IMR based HDD (Line 15).

Please be noted, in general, the *sync*-like function (e.g., *sync* [2, 7], *fdatasync* [3], *flush* [4]) is used to ensure that the data previously written by the *write*-like function(s) can be safely persisted into the storage device such as HDD. That is, the *sync*-like functions invoked during the naïve RMWs (i.e., Lines 7, 9 and 12) are actually served as critical *write barriers* to ensure that the correct sequence of “read-modify-write”, as described in [25], can be enforced. This also explains why for most of cases, there is no need to invoke a *sync*-like function after writing the $SST[i]$ into $TrackList[i]$ which does not incur any track rewrite(s) (i.e., Lines 13~14). However, if there exist un-synced track(s) which are adjacent to the to-be-written $TrackList[i]$, a *sync*-like function may

Algorithm 1: SSTable Persisting with Naïve RMW

```
Input: SST: the content of an SSTable to be persisted.
Input: TrackList: the list of the allocated tracks for SST.
1 for  $i \leftarrow 0$  to TrackList.size do
2   if any adjacent tracks of TrackList[ $i$ ] is un-synced then
3     sync; // ensure valid data are persisted
4   if writing to TrackList[ $i$ ] incurs top track rewrite(s) then
5     /* -- Naïve RMW Begin -- */
6     read the valid data from adjacent top track(s);
7     write the valid data into BackupRegion;
8     sync; // ensure valid data are persisted
9     write SST[ $i$ ] to TrackList[ $i$ ];
10    sync; // ensure SST[ $i$ ] is persisted
11    read the valid data from BackupRegion;
12    write back the valid data into adjacent top track(s);
13    sync; // ensure valid data are persisted
14    /* -- Naïve RMW End -- */
15  else
16    write SST[ $i$ ] to TrackList[ $i$ ];
17 sync; // ensure the entire SST is persisted
```

be invoked to ensure the valid data of un-synced tracks are persisted into HDD by a correct sequence (Lines 2~3).

3.3.2 SSTable Persisting with Merged RMW

Although the process of *SSTable Persisting with Naïve RMW* (i.e., Algorithm 1) looks simple and elegant, nevertheless, based on our investigation, there exist two sources of inefficiency hid behind it when the size of an SSTable is (much) larger than the size of an IMR track. First, although the *sync*-like functions ensure the *Naïve RMW* approach against unexpected crashes, they also bring severe performance degradation to the whole process. The rationale behind this is that the *sync*-like functions have adverse effects on I/O performance of HDD [2, 24, 45]. Second, since *Naïve RMW* approach must be performed on a *track-by-track* basis, the valid data of some top tracks are actually backed up and written back redundantly, resulting in the doubled read, write and *sync* workloads.

Thus, in order to further improve the efficiency of persisting the buffered SSTable from the *Write Buffer* into the allocated track(s) when the RMWs are incurred, this section introduces a novel *Merged RMW* approach. Its key idea is to *re-order* multiple *track-by-track* naïve RMWs into a single “merged RMW”, so as to significantly reduce the number of required *sync*-like functions and avoid redundant track rewrites while still ensure the crash consistency.

Algorithm 2 depicts the process of *SSTable Persisting with Merged RMW*, which persists the buffered SSTable (denoted as SST) from the *Write Buffer* into the allocated track(s) (denoted as *TrackList*) by employing the newly proposed *Merged RMW* approach. First of all, different from the process shown in Algorithm 1, this process firstly performs the

Algorithm 2: SSTable Persisting with Merged RMW

```
Input: SST: the content of an SSTable to be persisted.  
Input: TrackList: the list of the allocated tracks for SST.  
// The parts of SST incurring track rewrites  
1 if persisting SST incurs track rewrites then  
    /* -- Merged RMW Begin -- */  
2     for  $i \leftarrow 0$  to TrackList.size do  
3         if writing to TrackList[i] incurs track rewrites then  
4             read the valid data from adjacent top track(s);  
5             write the valid data to BackupRegion;  
6         sync; // ensure valid data are persisted  
7     for  $i \leftarrow 0$  to TrackList.size do  
8         if writing to TrackList[i] incurs track rewrites then  
9             write SST[i] to TrackList[i];  
10        sync; // ensure SST[i] is persisted  
11    for  $i \leftarrow 0$  to TrackList.size do  
12        if writing to TrackList[i] incurs track rewrites then  
13            read the valid data from BackupRegion;  
14            write back the valid data into adjacent top  
            track(s);  
15        sync; // ensure valid data are persisted  
    /* -- Merged RMW End -- */  
// The rest parts of SST w/o track rewrites  
16 for  $i \leftarrow 0$  to TrackList.size do  
17     if SST[i] is un-written && TrackList[i] is a bottom then  
18         write SST[i] to TrackList[i];  
19 sync; // ensure data to bottom are persisted  
20 for  $i \leftarrow 0$  to TrackList.size do  
21     if SST[i] is un-written && TrackList[i] is a top then  
22         write SST[i] to TrackList[i];  
23 sync; // ensure the entire SST is persisted
```

Merged RMW approach to handle all the allocated bottom track(s) that would incur top track rewrite(s) on a batch basis as follows (Lines 1~15): First, it backs up the valid data of all the involved adjacent top track(s) into the *BackupRegion* in a batch (Lines 2~5) before invoking the *first* sync-like function to ensure all of these valid data are safely persisted into the IMR based HDD (Line 6); Second, it writes the content of SST into all the involved bottom tracks in a batch (Lines 7~9) before invoking the *second* sync-like function to ensure these written parts of SST are safely persisted into the IMR based HDD (Line 10); Thirdly, it moves back all the backed-up valid data from the *BackupRegion* into the involved adjacent top track(s) in a batch (Lines 11~14) before invoking the *third* sync-like function to ensure these backed-up valid data are safely persisted into the IMR based HDD (Line 15).

Next, this process persists the remaining parts of SST, which would not incur top track rewrite(s), into the corresponding tracks (Lines 16~23) as follows: First, this process gives the highest priority to write the un-written parts

of SST into their corresponding bottom tracks in a batch (Lines 16~18). This is because, if the un-written parts of SST are written into top tracks first, the subsequent writes to adjacent bottom tracks may instead incur extra track rewrites over those “just-written” top tracks. Then, a sync-like function should be invoked (Line 19) to ensure the data writes to bottom tracks (if any) are persisted into HDD. Finally, this process writes the un-written parts of SST into their corresponding top tracks in a batch, followed by invoking a sync-like function to ensure the entire SST are safely persisted into HDD (Lines 20~23).

Compared with the SSTable persisting process with *Naïve RMW* approach (i.e, Algorithm 1), the SSTable persisting process with the *Merged RMW* approach not only significantly minimizes the number of required sync-like functions (specifically, at least 1 and at most 5), but also avoids the redundant backup and write-back of top tracks (since the involved top tracks are only backed up and written back once). Moreover, the *Merged RMW* approach nicely ensures the crash consistency for the SSTable by backing up the parts of SSTable residing in top tracks into the *BackupRegion*, before writing any data into bottom tracks that would incur top track rewrites. However, it is worth noting that the *Merged RMW* approach requires a larger *BackupRegion* than the *Naïve RMW* approach, in order to back up the valid data of all the involved adjacent top track(s) in a batch. Specifically, the size of the *BackupRegion* required by the *Merged RMW* approach is just twice as the size of an SSTable, which should be a negligible storage overhead compared with the total capacity of HDD.

Notably, as astute readers may have noticed, an alternative approach, which organizes all tracks into “regions” of a fixed number of consecutive tracks, is also able to reduce the numbers of track rewrites and sync-like functions as does KVIMR. That is, this approach may always write a whole SSTable into a fixed-sized region by writing all the bottom tracks then all the top tracks in the region. Although such approach looks simple and effective indeed, however, we argue that it might inevitably waste the disk space (since the actual sizes of SSTables are not fixed and could be smaller than the fixed SSTable size limit in the mainstream LSM-tree based KV store implementations). Thus, KVIMR adopts a more flexible *Compaction-aware Track Allocation* and *Merged RMW* to avoid such space waste issue while reduce the numbers of track rewrites and sync-like functions.

3.4 Crash Consistency

Since the proposed KVIMR does not introduce significant changes to the core design of LSM-tree based KV stores, the crash consistency mechanisms of different LSM-tree based KV stores still remain and work nicely in the same way.

KVIMR further ensures the crash consistency regarding its key metadata (i.e., SSTable-to-Track Map (S2TMap)) as follows: First, to avoid incurring the time-consuming RMW

processes over IMR tracks, we propose to persist the track-level S2TMap into a few reserved top tracks of IMR based HDD or other persistent storage device, which can be freely updated, in the system. Second, to better manage the persisting overhead, we propose to persist S2TMap whenever the LSM-tree based KV store persists their key metadata (e.g., the *manifest* file) after each successful compaction process. This ensures that the metadata of KVIMR can always be consistent with that of the LSM-tree based KV store, so that both LSM-tree based KV store and KVIMR can be recovered to a consistent state successfully after unexpected system crashes.

4 Evaluation

4.1 Evaluation Setup

In this section, we evaluate the effectiveness of the proposed KVIMR middleware. Since there is no available real products of IMR based HDDs to date, we follow the emulation approach suggested in [39] to emulate an 100 GB IMR based HDD with a real CMR based HDD (model no. ST500DM002 [1]) so that the evaluated results can accurately reflect actual performance of the disk internal activities. Particularly, we split the LBA range into 2 MB tracks as suggested in [25, 50], organize the tracks by the interlaced track layout of IMR, and read/write to tracks with IMR-like restrictions [28]. In addition, we architect the KVIMR as a middleware as introduced in Section 3.1 and develop KVIMR in the user-space using C++, and employ the existing `pread/pwrite/fdatasync` system calls with the `O_DIRECT` flag to directly manage SSTables and perform the RMW approach over tracks of the emulated IMR based HDD. Moreover, the following schemes are implemented and integrated with the KVIMR middleware for evaluation purposes:

- **Seq** allocates all tracks in a sequential order (i.e., one track followed by the subsequent one(s)) but skips tracks that have been allocated by other SSTables; and it adopts the *Naïve RMW* approach [25] to perform track rewrites.
- **3Phase** allocates the unused tracks in the following order: bottom tracks, every other top tracks, and the rest of top tracks [19, 20]; and it adopts the *Naïve RMW* approach [25] to perform track rewrites.
- **KVIMR-N** denotes the proposed *Compaction Aware Track Allocation* scheme (presented in Section 3.2) with the *Naïve RMW* approach [25].
- **KVIMR-M** denotes the proposed *Compaction Aware Track Allocation* scheme (presented in Section 3.2) with the proposed *Merged RMW* approach (presented in Section 3.3).

On the other hand, we modify three well-known implementations of LSM-tree based KV store, i.e., RocksDB [18], LevelDB [21], and HyperLevelDB [17], so that they can access its SSTables through a set of POSIX complaint file operations provided by the KVIMR middleware as presented in Section 3.1. The metadata (e.g., *manifest*, *WAL*) of these

LSM-tree based KV store implementations are currently kept in an SSD alongside (like GearDB’s implementation [53]). However, since the size of metadata is relatively small, these metadata can also be managed in a few IMR tracks with little performance impact. Notably, because of limited page length, we mainly demonstrate the evaluation results collected from RocksDB. In addition, we adopt the default settings [18] (e.g., the SSTable size is 64 MB, the size limit of level 1 is 256 MB, `kLO_SlowdownWritesTrigger` and `kLO_StopWritesTrigger` are 20 and 36 respectively) to configure RocksDB, but further set the `compaction_readahead_size` to 2 MB (as suggested in [6]) and `bloom_bits` to 10 (as suggested in [6]) to have better performance on HDD. Moreover, we revise the benchmark tool `db_bench` released with RocksDB, LevelDB, and HyperLevelDB, so as to evaluate the performance of different schemes under various workloads generated by YCSB [14]. All the experiments are conducted on a workstation PC, which is equipped with two Intel(R) Xeon(R) E5-1630 v4 @ 3.70 GHz processors and 16 GB DDR4 DIMM memory, where the operating system is 64-bit Ubuntu 14.04.1 LTS with Linux kernel version 3.13.0.

4.2 Evaluation Results

4.2.1 Overall Load Performance

This section investigates the overall performance under different schemes by randomly loading/inserting 75 millions of 1 KB KV pairs into RocksDB. Figure 5 shows the overall performance results, where the x-axis of each sub-figure denotes different schemes and the y-axis of each sub-figure shows the results from different performance metrics. For ease of the comparison, the throughput of adopting three-phase allocation scheme to allocate CMR tracks for accommodating SSTables (denoted as CMR as in Section 2.3) is also plotted in Figure 5 as a horizontal dashed line. From Figure 5a, we can firstly observe that both KVIMR-N and KVIMR-M achieve significant throughput improvements when compared with Seq and 3Phase. Specifically, KVIMR-N achieves $1.44\times$ and $2.21\times$ higher throughput than that of Seq and 3Phase. Moreover, it can be also observed that KVIMR-M further improves the throughput by 7.0% as compared to KVIMR-N, because of the improved compaction efficiency contributed by the proposed Merged RMW approach. More encouragingly, KVIMR-M achieves a comparable throughput as compared to CMR with only about 5.5% degradation.

To better demonstrate the compaction efficiency of different schemes, Figures 5b and 5c respectively demonstrate the number of compactions and the sum of execution time of compactions (or *cumulative compaction time* [18]) during the loading process. It can be clearly observed from Figure 5b that all the schemes share similar number of compactions. This is because all the schemes are implemented as middleware (similar to the proposed KVIMR), and the design regarding how RocksDB performs compaction processes

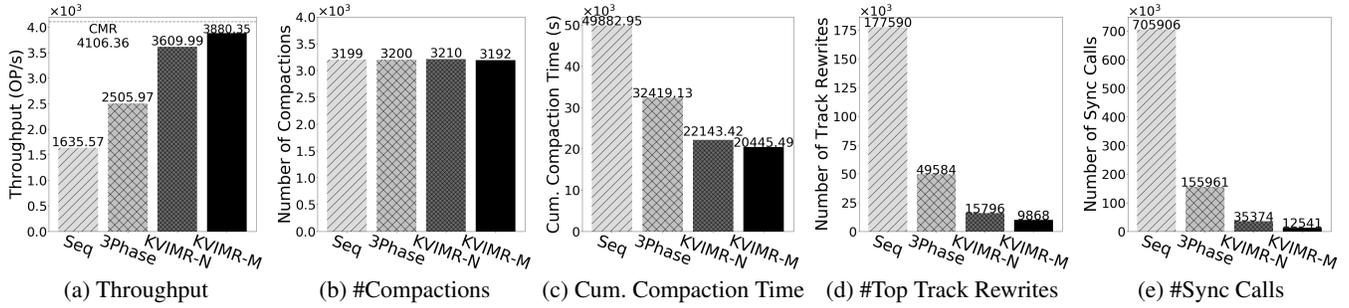


Figure 5: Overall Performance Results of Loading 75 Millions of KV Pairs into RocksDB.

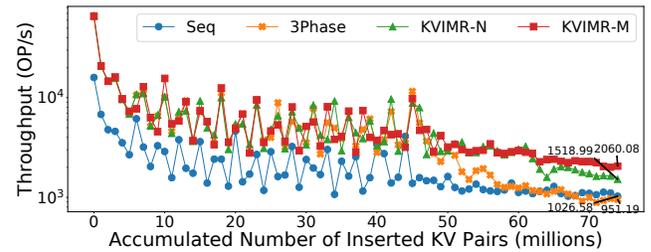
is not changed. However, from Figure 5c, we can observe that, KVIMR-N and KVIMR-M effectively reduce the cumulative compaction time required to complete a similar numbers of compactions, and thereby greatly improve the compaction efficiency (i.e., the average time required to perform a compaction process). Specifically, KVIMR-N largely reduces the cumulative compaction time by 55.61% and 31.69% as compared to Seq and 3Phase. Moreover, KVIMR-M further reduces the cumulative compaction execution time by 7.67% as compared to KVIMR-N.

Based on our investigation, the improvement on the overall throughput and the reduction on the cumulative compaction time (achieved by KVIMR-N and KVIMR-M) can be mainly attributed to the prevention of time-consuming RMW processes, which may incur additional track rewrites and sync calls to slow down the efficiency of persisting SSTables. Figures 5d and 5e reveal the numbers of (top) track rewrites and the numbers of sync calls incurred by different schemes during the loading process. It can be firstly observed that, KVIMR-N and KVIMR-M significantly reduce the numbers of top track rewrites and sync calls as compared to Seq and 3Phase. Specifically, KVIMR-N decreases the number of top track rewrites by 91.11% and 68.14% and diminishes the number of sync calls by 94.99% and 77.32% as compared to Seq and 3Phase, respectively. The main reason behind such reductions is that, KVIMR-N leverages the *compaction frequency* to allocate the IMR tracks for accommodating SSTables, so as to minimize the occurrence of time-consuming RMW processes. It is also worthy to see that, compared to KVIMR-N, KVIMR-M further reduces the numbers of top track rewrites and sync calls by 37.53% and 64.55%. The is because KVIMR-M adopts the proposed Merged RMW to cleverly circumvent the redundant top track rewrites and bound the number of sync calls, when the time-consuming RMWs cannot be fully avoided by the proposed Compaction-aware Track Allocation.

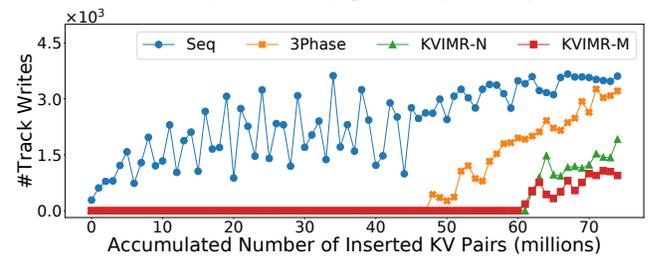
4.2.2 Load Performance Changes

Besides the overall performance results presented in Figure 5, it is also valuable to investigate how the throughput and other performance metrics actually change along the whole loading process. Figures 6a, 6b and 6c respectively show the throughput, the number of incurred track rewrites, and the number of invoked sync calls for loading every 1 million of KV pairs,

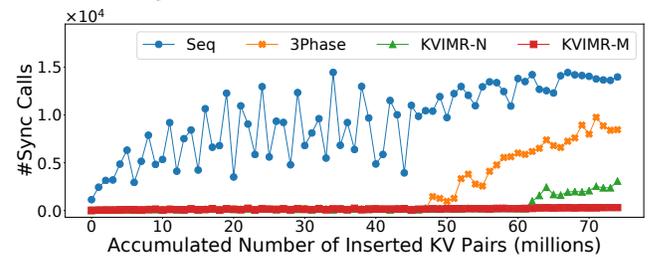
where the x-axis of each sub-figure denotes the accumulated number of inserted/loaded KV pairs.



(a) Changes in Throughput during Loading.



(b) Changes in the Number of Incurred Track Rewrites.



(c) Changes in the Number of Invoked Sync Calls.

Figure 6: Performance Changes during Loading.

First of all, from Figure 6a, we can observe that, compared with Seq and 3Phase, KVIMR-N and KVIMR-M lead to the much higher throughputs, almost for every 1 million of inserted KV pairs along the whole loading process. The main reason behind this is that, as revealed by Figures 6b and 6c, KVIMR-N and KVIMR-M incur much less number of top track rewrites and sync calls during the whole loading process. Interestingly, after inserting about 60 millions of KV pairs, KVIMR-M starts to achieve the highest throughput than the rest of schemes. For example, KVIMR-M achieves 26.27% higher throughput than that of KVIMR-N for the last 1 million of in-

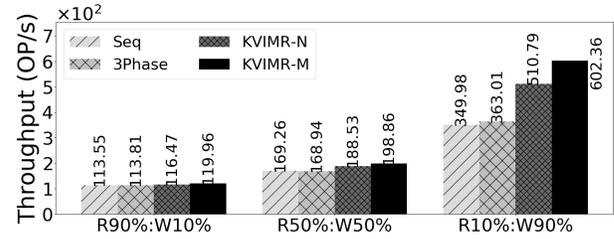
serted KV pairs. The reason is that, when the RMW processes are inevitable during the late loading process, the proposed Merged RMW approach can cleverly avoid redundant top track rewrites and bound the number of sync calls, resulting in the improved efficiency of persisting SSTables.

By contrast, *Seq* achieves the lowest throughput almost during the whole loading process, and *3Phase* starts to encounter noticeable throughput degradation after inserting about 50 millions of KV pairs and suffers very low throughput as *Seq* at the very end of the loading process. In particular, as observed from Figure 6a, *KVIMR-M* achieves $2.01\times$ and $2.17\times$ higher throughput than that of *Seq* and *3Phase* respectively for the last 1 million of inserted KV pairs. Such serious throughput degradations of *Seq* and *3Phase* can be attributed to their track allocation designs. Particularly, *Seq* writes the SSTable data in a strictly-sequential order (i.e., one track followed by the subsequent one(s)), resulting in that the time-consuming RMW processes are mostly incurred whenever writing to any bottom track(s). On the other hand, *3Phase* only utilizes bottom tracks to accommodate the SSTable data and fully avoids RMW processes during the first phase (i.e., before inserting 50 millions of KV pairs). However, when *3Phase* starts to allocate top tracks to accommodate SSTables afterwards, it may incur more and more RMW processes as the space usage of HDD increases due to the lack of consideration regarding compaction frequency.

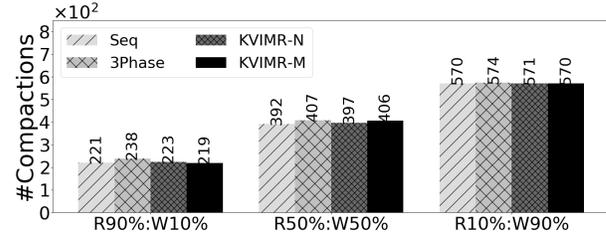
4.2.3 Throughput under Different Workloads

To further investigate the throughput under read-write mixed workloads, we randomly insert and retrieve 7.5 millions of KV pairs after loading the 75 millions of KV pairs into RocksDB. In particular, three workloads of different read/write (R/W) ratios (i.e., R90%:W10%, R50%:W50%, and R10%:W90%) generated by YCSB [14], which follow the Zipfian distribution, are used to represent different *write intensities*. Figure 7a shows the throughputs of different schemes, where the x-axis denotes different R/W ratios and the y-axis shows the throughput. It can be firstly observed that the throughputs of all the evaluated schemes increase as the write intensity (or write ratio) increases. This is because, randomly reading a large amount of KV pairs from LSM-tree based KV store might seriously slow down the overall throughput due to the poor random access performance of HDD.

More importantly, as revealed by Figure 7a, *KVIMR-N* and *KVIMR-M* are more effective in improving the throughput as the write intensity keeps increasing, as compared to *Seq* and *3Phase*. Specifically, *KVIMR-N* and *KVIMR-M* lead to at least 2.28%, 10.22% and 28.93% throughput improvements (than that of *Seq* and *3Phase*) under workloads of R90%:W10%, R50%:W50% and R10%:W90% respectively. This is because, as shown in Figure 7b, for all the evaluated schemes, the number of incurred compactions increases as the write intensity increases. Such increasing number of compactions implies that more SSTables need to be persisted into HDD, leaving a



(a) Throughput under Workloads of Various R/W Ratios.



(b) #Compactions under Workloads of Various R/W Ratios.

Figure 7: Throughput under Workloads of Various R/W Ratios

larger room for *KVIMR-N* and *KVIMR-M* to reduce the number of track rewrites and sync calls for higher throughput gains (under workloads of higher write intensities).

4.2.4 Throughput under Different Sizes of SSTable

To understand how the impact of SSTable size on the throughput, we randomly loading/inserting 75 millions of 1 KB KV pairs into RocksDB under different sizes of SSTable. Specifically, as suggested in the [8], since HDD typically demonstrates better performance under larger sizes of SSTable, this section mainly evaluates the throughput of different schemes when SSTable size ranges from 64MB to 512MB.

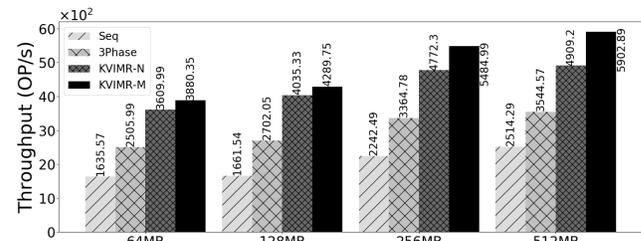


Figure 8: Throughput under Different Sizes of SSTable.

Figure 8 shows the overall load performance results, where the x-axis denotes different SSTable sizes and the y-axis denotes the throughput. It can be firstly observed that all the schemes achieve higher throughput with larger SSTable sizes. Besides, *KVIMR* still effectively and stably achieves better throughput than that of *Seq* and *3Phase* as the SSTable size increases. More interestingly, as compared to *KVIMR-N*, *KVIMR-M* tends to be more effective in improving the throughput when the SSTable size increases. Specifically, the performance gaps between *KVIMR-N* and *KVIMR-M* are 7.0%, 5.9%, 13.0%, 16.8% with SSTables sizes of 64 MB, 128 MB, 256 MB and 512 MB respectively. The key reason of such increasing performance gap is that the Merged RMW adopted

by KVIMR-M has potential to circumvent more redundant top track rewrites and reduce more number of sync calls with larger SSTable sizes.

4.2.5 Support for Other LSM-tree based KV Stores

To demonstrate the great compatibility of KVIMR, we further conduct the load throughput evaluation (i.e., loading 75 millions of 1 KB KV pairs generated by YCSB [14]) on other two well-known LSM-tree based KV stores: LevelDB [21], and HyperLevelDB [17]. Notably, to present the performance results on the same basis, we apply the same configurations adopted by RocksDB to both LevelDB and HyperLevelDB for achieving better HDD performance. That is, we set the SSTable size to 64 MB, set the size limit of level 1 to 256 MB and set *kLO_SlowdownWritesTrigger* and *kLO_StopWritesTrigger* to 20 and 36 respectively. Moreover, since LevelDB and HyperLevelDB lack the design of *compaction_readahead_size*, we instead set the *block_size* to 2 MB to have better performance on HDD as suggested by [6].

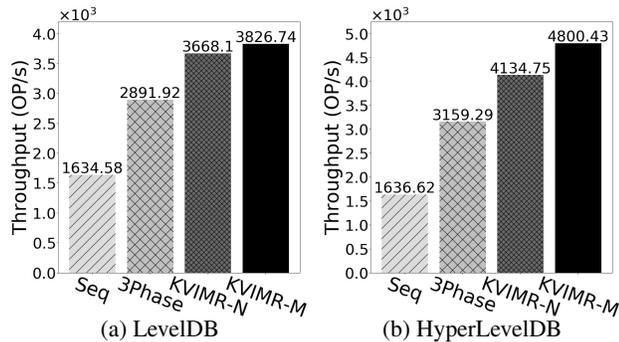


Figure 9: Throughput of Loading 75 Millions of KV Pairs into Other Well-Known LSM-tree based KV Stores.

Figures 9a and 9b show the overall throughput achieved by LevelDB and HyperLevelDB respectively upon loading 75 millions of 1 KB KV pairs. It can be clearly observed that the trends of overall throughputs achieved by the evaluated schemes under LevelDB and HyperLevelDB are quite similar to what we can observe under RocksDB (see Figure 5a). That is, KVIMR-N and KVIMR-M still effectively achieve better throughputs than that of Seq and 3Phase by at least 55.43% and 21.16% under LevelDB respectively (and by at least 60.41% and 23.59% under HyperLevelDB respectively).

5 Related Work

There exist some mentionable studies proposed to address the track rewrite issue of Shingled Magnetic Recording (SMR) based HDD by introducing the following techniques to the existing LSM-tree based KV store implementations. For example, SMRDB [39] proposes to enlarge the SSTable size to the band/zone size of SMR based HDD to avoid track rewrites, arranges SSTables into only two levels with the key ranges of SSTables overlapped within the same level, and presents a new cost-considered compaction design. Another study called

SEALDB [52] proposes to group the SSTables involved in a compaction into a set, and then sequentially writes a set of SSTables into a variable-sized *dynamic band* to mitigate the track rewrite issue. A more recent study namely GearDB [53] proposes to sequentially write the SSTables of the same level into the same *SMR band/zone*, and further introduces a *Gear Compaction* to avoid garbage collection in SMR based HDD.

The aforementioned designs indeed take effect at mitigating the track rewrite issue of SMR; however, they could be *ineffective* when blindly applied to IMR based HDD since SMR and IMR technologies adopt naturally-different track layouts. Specifically, all these designs must write the SSTables into SMR tracks in a “strictly-sequential” order (i.e., one track followed by the subsequent one(s)). As revealed by our evaluation, sequentially writing SSTables into IMR tracks may incur serious throughput degradation, since writing to any bottom track may cause the time-consuming RMW(s) to rewrite its adjacent top track(s) in the IMR based HDD. On the other hand, all these designs are developed by revamping the existing LSM-tree based KV store (i.e., LevelDB [21]). That is, they are all tightly-coupled with one specific implementation of the LSM-tree based KV store, making them costly to be upgraded with the software updates of KV store and integrated with other representative implementations of the LSM-tree based KV store, such as the widely deployed RocksDB [18].

6 Conclusion

This paper presents KVIMR, a data management middleware, to construct a cost-effective yet high-throughput LSM-tree based KV store on IMR based HDD. KVIMR delivers great compatibility for mainstream LSM-tree based KV store implementations without introducing significant modifications by being architected as a middleware sitting between LSM-tree based KV store and IMR based HDD. Technically, KVIMR remedies the throughput degradation resulted from IMR through the proposing of two novel designs: a *Compaction Aware Track Allocation* scheme to minimize the time-consuming RMWs and efficiently access the SSTables during the compaction process, and a *Merged RMW* approach to improve the efficiency of persisting an SSTable into IMR based HDD when the time-consuming RMWs are inevitable. Our evaluations on three well-known LSM-tree based KV store implementations (i.e., RocksDB, LevelDB, and HyperLevelDB) reveal that KVIMR not only improves the overall throughput by up to $1.55\times$ under write-intensive workloads but even achieves $2.17\times$ higher throughput under high space usage of HDD, as compared with the state-of-the-art three-phase track allocation scheme for IMR.

7 Acknowledgements

We thank our shepherd, Raja Appuswamy, and all the anonymous reviewers for their valuable comments and suggestions. This work is supported in part by The Research Grants Council of Hong Kong SAR (Project No. CUHK24209618).

References

- [1] Desktop hdd product manual standard models st1000dm003 st500dm002. <https://www.seagate.com/www-content/product-content/desktop-hdd-fam/en-us/docs/100768625b.pdf>.
- [2] Ensuring data reaches disk. <https://lwn.net/Articles/457667/>.
- [3] fdatsync. <https://man7.org/linux/man-pages/man2/fdatasync.2.html>.
- [4] Hgst. libzbc version 5.4.1. <https://github.com/hgst/libzbc>.
- [5] POSIX: Portable Operating System Interface. <https://pubs.opengroup.org/onlinepubs/9699919799/>.
- [6] Rocksdb tuning guide. <https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide>.
- [7] Sync. <https://man7.org/linux/man-pages/man2/sync.2.html>.
- [8] Tuning rocksdb on spinning disks. <https://github.com/facebook/rocksdb/wiki/Tuning-RocksDB-on-Spinning-Disks>.
- [9] Write stalls. <https://github.com/facebook/rocksdb/wiki/Write-Stalls>.
- [10] Abutalib Aghayev, Mansour Shafaei, and Peter Desnoyers. Skylight—a window on shingled disk operation. *ACM Transactions on Storage (TOS)*, 11(4):16, 2015.
- [11] Ahmed Amer, JoAnne Holliday, Darrell DE Long, Ethan L Miller, Jehan-François Pâris, and Thomas Schwarz. Data management and layout for shingled magnetic recording. *IEEE Transactions on Magnetics*, 47(10):3691–3697, 2011.
- [12] W. A. Challener, C. Peng, A. V. Itagi, D. Karns, Y. Peng, X. Yang, X. Zhu, N. J. Gokemeijer, Y. T. Hsia, G. Ju, R. E. Rottmayer, M. A. Seigler, and E. C. Gage. The road to hamr. In *Magnetic Recording Conference, 2009. APMRC '09. Asia-Pacific*, pages 1–2, Jan 2009.
- [13] 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 (TOCS)*, 26(2):4, 2008.
- [14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Yahoo! cloud serving benchmark (ycsb), 2010.
- [15] 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 *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.
- [16] E. A. Dobisz, Z. Z. Bandic, T. W. Wu, and T. Albrecht. Patterned media: Nanofabrication challenges of future disk drives. *Proceedings of the IEEE*, 96(11):1836–1846, Nov 2008.
- [17] Robert Escriva, Sanjay Ghemawat, David Grogan, Jeremy Fitzhardinge, and Chris Mumford. Hyperleveldb, 2019. <https://github.com/rescrv/HyperLevelDB>.
- [18] Facebook. Rocksdb: a persistent key-value store for fast storage environments., 2011. <https://github.com/facebook/rocksdb>.
- [19] Kaizhong Gao, Wenzhong Zhu, and Edward Gage. Write management for interlaced magnetic recording devices, November 29 2016. US Patent 9,508,362.
- [20] Kaizhong Gao, Wenzhong Zhu, and Edward Gage. Interlaced magnetic recording, August 8 2017. US Patent 9,728,206.
- [21] Sanjay Ghemawat and Jeff Dean. Leveldb, 2011. <https://github.com/google/leveldb>.
- [22] Steven Granz, Jason Jury, Chris Rea, Ganping Ju, Jan-Ulrich Thiele, Tim Rausch, and Edward Gage. Areal density comparison between conventional, shingled, and interlaced heat-assisted magnetic recording with multiple sensor magnetic recording. *IEEE Transactions on Magnetics*, PP:1–3, 09 2018.
- [23] Steven Granz, Wenzhong Zhu, Edmun Chian Song Seng, Utt Heng Kan, Chris Rea, Ganping Ju, Jan-Ulrich Thiele, Tim Rausch, and Edward C Gage. Heat-assisted interlaced magnetic recording. *IEEE Transactions on Magnetics*, 54(2):1–4, 2017.
- [24] Jorge Guerra, Leonardo Mármol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. Software persistent memory. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 319–331, 2012.
- [25] Mohammad Hossein Hajkazemi, Ajay Narayan Kulkarni, Peter Desnoyers, and Timothy R Feldman. Track-based translation layers for interlaced magnetic recording. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 821–832, 2019.

- [26] Tyler Harter, Dhruva Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis of {HDFS} under hbase: A facebook messages case study. In *Proceedings of the 12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)*, pages 199–212, 2014.
- [27] Weiping He and David HC Du. Smart: An approach to shingled magnetic recording translation. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 121–134, 2017.
- [28] Euiseok Hwang, Jongseung Park, Richard Rauschmayer, and Bruce Wilson. Interlaced magnetic recording. *IEEE Transactions on Magnetics*, 53(4):1–7, 2016.
- [29] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *6th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.
- [30] P. Kasiraj, R. New, J. de Souza, and M. Williams. System and method for writing data to dedicated bands of a hard disk drive. In *US Patent 7490212*, Feb 2009.
- [31] A. Kikitsu, Y. Kamata, M. Sakurai, and K. Naito. Recent progress of patterned media. *IEEE Transactions on Magnetics*, 43(9):3685–3688, Sept 2007.
- [32] M. H. Kryder, E. C. Gage, T. W. McDaniel, W. A. Challenor, R. E. Rottmayer, G. Ju, Y. T. Hsia, and M. F. Erden. Heat assisted magnetic recording. *Proceedings of the IEEE*, 96(11):1810–1835, Nov 2008.
- [33] 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 *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2015.
- [34] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.
- [35] Shaoping Li, Gerardo A Bertero, Michael L Mallary, Ge Yi, and Steven C Rudy. Connection schemes for a multiple sensor array usable in two-dimensional magnetic recording, November 18 2014. US Patent 8,891,207.
- [36] Lanyue Lu, Thanumalayan Sankaranarayanan Pillai, Harisharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):5, 2017.
- [37] Miha Marolt and Z Jaglicic. Superparamagnetic materials. In *Proceeding of Seminar 1b-4th Year (Old Program)*, University of Ljubljana Faculty of Mathematics and Physics, 2014.
- [38] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.
- [39] Rekha Pitchumani, James Hughes, and Ethan L Miller. Smrdb: key-value data store for shingled magnetic recording disks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 18. ACM, 2015.
- [40] HJ Richter, AY Dobin, RT Lynch, D Weller, RM Brockie, O Heinonen, KZ Gao, J Xue, RJM vd Veerdonk, P Asselin, et al. Recording potential of bit-patterned media. *Applied Physics Letters*, 88(22):222512, 2006.
- [41] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.
- [42] Mansour Shafaei, Mohammad Hossein Hajkazemi, Peter Desnoyers, and Abutalib Aghayev. Modeling drive-managed smr performance. *ACM Transactions on Storage (TOS)*, 13(4):38, 2017.
- [43] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 18–18. USENIX Association, 2012.
- [44] I. Tagawa and M. Williams. High density data-storage using shingled-write. In *IEEE International Magnetic Conference*, 2009.
- [45] Shucheng Wang, Ziyi Lu, Qiang Cao, Hong Jiang, Jie Yao, Yuanyuan Dong, and Puyuan Yang. {BCW}: Buffer-controlled writes to hdds for ssd-hdd hybrid storage server. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 253–266, 2020.
- [46] R. Wood, M. Williams, A. Kavcic, and J. Miles. The feasibility of magnetic recording at 10 terabits per square inch on conventional media. *IEEE Transactions on Magnetics*, 45(2):917–923, Feb 2009.
- [47] Roger Wood. Shingled magnetic recording and two-dimensional magnetic recording. *IEEE Magnetics Society, Santa Clara Valley*, 2010.
- [48] Fenggang Wu, Bingzhe Li, Baoquan Zhang, Zhichao Cao, Jim Diehl, Hao Wen, and David HC Du. Tracklace:

Data management for interlaced magnetic recording. *IEEE Transactions on Computers*, 2020.

- [49] Fenggang Wu, Ming-Chang Yang, Ziqi Fan, Baoquan Zhang, Xiongzi Ge, and David HC Du. Evaluating host aware {SMR} drives. In *8th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.
- [50] Fenggang Wu, Baoquan Zhang, Zhichao Cao, Hao Wen, Bingzhe Li, Jim Diehl, Guohua Wang, and David HC Du. Data management design for interlaced magnetic recording. In *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.
- [51] Ming-Chang Yang, Yuan-Hao Chang, Fenggang Wu, Tei-Wei Kuo, and David HC Du. On improving the write responsiveness for host-aware smr drives. *IEEE Transactions on Computers*, 68(1):111–124, 2018.
- [52] Ting Yao, Zhihu Tan, Jiguang Wan, Ping Huang, Yiwen Zhang, Changsheng Xie, and Xubin He. Sealdb: An efficient lsm-tree based kv store on smr drives with sets and dynamic bands. *IEEE Transactions on Parallel and Distributed Systems*, 30(11):2595–2607, 2019.
- [53] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. Geardb: a gc-free key-value store on hm-smr drives with gear compaction. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 159–171, 2019.
- [54] Hwanjin Yong, Kisik Jeong, Joonwon Lee, and Jin-Soo Kim. vstream: virtual stream management for multi-streamed ssds. In *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.
- [55] J. G. Zhu, X. Zhu, and Y. Tang. Microwave assisted magnetic recording. *IEEE Transactions on Magnetics*, 44(1):125–131, Jan 2008.

Differentiated Key-Value Storage Management for Balanced I/O Performance

Yongkun Li¹, Zhen Liu¹, Patrick P. C. Lee², Jiayu Wu¹, Yinlong Xu^{1,3}
Yi Wu⁴, Liu Tang⁴, Qi Liu⁴, Qiu Cui⁴

¹University of Science and Technology of China ²The Chinese University of Hong Kong
³Anhui Province Key Laboratory of High Performance Computing, USTC ⁴PingCAP

Abstract

Modern key-value (KV) stores adopt the LSM-tree as the core data structure for managing KV pairs, but suffer from high write and read amplifications. Existing LSM-tree optimizations often make design trade-offs and are unable to simultaneously achieve high performance in writes, reads, and scans. To resolve the design tensions, we propose DiffKV, which builds on KV separation to carefully manage the ordering for keys and values. DiffKV manages keys using the conventional LSM-tree with *fully-sorted ordering* (within each level of the LSM-tree), while managing values with *partially-sorted ordering* with respect to the fully-sorted ordering of keys in a coordinated way for preserving high scan performance. We further propose fine-grained KV separation to differentiate KV pairs by size, so as to realize balanced performance under mixed workloads. Experimental results show that DiffKV can simultaneously achieve the best performance in all aspects among existing LSM-tree-optimized KV stores.

1 Introduction

Key-value (KV) storage, which abstracts data as KV pairs, becomes a critical storage paradigm for supporting a variety of applications, such as search engines [4, 11, 23, 53], distributed file systems [1, 28], data deduplication [18, 41], graph stores [7, 22, 37, 64], and OLTP/OLAP databases [3, 8, 11, 19, 30, 31, 33, 43, 50, 65]. Such storage paradigms mainly provide three operations: (i) *writes*, which insert KV pairs, (ii) *reads*, which retrieve the value of a single key, and (iii) *scans*, which retrieve the values over a key range. In particular, scans are essential operations for KV stores in various applications; for example, graph computation tasks may traverse multiples nodes or edges [2, 7, 37], and OLAP databases traverse multiple entries in a table for data filtering or aggregation [8, 30, 33, 50]. Thus, optimizations for scans have also been specifically studied in the literature [43, 55, 62, 65].

To better leverage the efficiency of sequential I/Os and preserve the data ordering for fast scans, modern KV stores (e.g., [10, 24, 27, 42, 52]) often adopt the *Log-Structured-Merge-tree (LSM-tree)* [46]. At a high level, the LSM-tree builds on three properties (see §2.1 for details): (i) it organizes KV pairs in a log-structured layout, in which updating KV pairs is treated as issuing sequential writes of KV pairs to persistent storage with high performance; (ii) it adopts a

multi-level structure that keeps KV pairs *fully sorted* within each level, so as to support efficient reads and scans without specialized in-memory index structures; and (iii) it mitigates the write overhead by gradually moving KV pairs from lower levels to higher levels via *compaction*, so as to maintain high storage scalability. However, the compaction incurs substantial amounts of I/Os, as the KV pairs in adjacent levels need to be retrieved and written back to maintain sorted ordering. It is well known that LSM-tree KV stores suffer from severe write and read amplifications, especially when the number of levels of the LSM-tree increases with the growing volume of KV pairs being managed [42, 52, 58].

To reduce the compaction overhead, a number of LSM-tree optimization techniques have been proposed in the literature. One direction of work is to relax the fully-sorted nature in each level of the LSM-tree, thereby alleviating the compaction overhead [17, 52, 55]. However, as the degree of fully-sorted ordering is relaxed, the scan performance also degrades. Another direction of work is based on *KV separation*, which separates the storage of keys and values by keeping only keys (in fully-sorted ordering) in the LSM-tree and performing value management in a dedicated storage area [10, 20, 38, 42, 49, 51, 63]. KV separation alleviates the compaction overhead as the LSM-tree size now significantly decreases without storing the values, and is particularly suited for practical KV workloads whose KV pairs are composed of large-size values [5, 9, 30, 38, 44, 59] (see §2.2 for details). However, it degrades the scan performance, especially for the values with small-to-medium sizes that are also common in practice [5, 9], since each scan needs to issue random I/Os to the values over a key range that are no longer fully sorted (in contrast, the random I/O overhead is amortized for large values). Also, KV separation incurs extra garbage collection (GC) overhead [10], thereby triggering additional I/O overhead beyond the compaction in the LSM-tree. In short, existing LSM-tree optimizations are still limited by tight performance tensions between reads/writes and scans, in terms of (i) the degrees of ordering in keys and values, and (ii) the management of KV pairs of varying sizes.

To this end, we design a novel KV store, DiffKV, that realizes balanced I/O performance on commodity storage devices (e.g., solid-state drives (SSDs)). Its main idea builds on the differentiated KV management in two aspects. First, DiffKV

differentiates the storage management of keys and values as in conventional KV separation, and takes one step further to carefully coordinate the differentiated management of the ordering for keys and values. Specifically, it keeps keys fully sorted in each level of the LSM-tree as in conventional KV separation for fast reads and scans, while keeping values *partially sorted*, such that the (partially-sorted) ordering of values is coordinated with respect to the (fully-sorted) ordering of keys. We design a new LSM-tree-like structure called the *vTree* for value management, such that the sorting of values in the *vTree* is triggered by the compaction of the LSM-tree. In this way, we limit the overhead of sorting values, yet we still maintain high scan performance via the partially-sorted ordering for values. Second, DiffKV differentiates the management of values via fine-grained KV separation, such that the KV pairs of different size groups are specifically managed for maintaining balanced performance under mixed workloads.

To the best of our knowledge, this is the first work that examines the differentiated ordering for KV pairs. Traditional LSM-tree KV stores without KV separation always couple keys and values together and only realize a single kind of ordering [24, 27, 52]. While existing KV separation designs [10, 20, 38, 42, 49, 51, 63] decouple keys and values, the values are unsorted, and they cannot be tuned to realize different degrees of ordering for balanced performance. Our main contributions are summarized as follows.

- We present DiffKV, which coordinates the differentiated management of ordering for keys and values, so as to simultaneously improve the performance of writes, reads, and scans. Specifically, DiffKV manages values in the *vTree* structure for the partially-sorted ordering of values.
- We propose multiple merge optimization techniques to reduce the sorting overhead in the *vTree*, and also develop a state-aware lazy GC scheme to realize high space efficiency and high performance.
- We propose fine-grained KV separation and differentiate the management of small, medium, and large KV pairs for optimizing mixed workloads. In addition to the LSM-tree and the *vTree*, we also propose a hotness-aware multi-log design for efficiently managing large KV pairs.
- We implement a DiffKV prototype atop Titan [51], an open-source KV store that implements KV separation with optimized techniques. Evaluation results show that DiffKV achieves the best performance in writes, reads, scans, and space utilization compared to state-of-the-art KV stores, including RocksDB [24], PebblesDB [52], and Titan [51].

We release the source code of our DiffKV prototype at <https://github.com/ustcadsl/diffkv>.

2 Background and Motivation

We first introduce the basics of an LSM-tree KV store. We then discuss the strengths and weaknesses of different LSM-tree optimizations to motivate our DiffKV design.

2.1 LSM-tree KV Store

Storage structure. Figure 1(a) depicts a simplified storage structure of a conventional LSM-tree KV store (e.g., LevelDB [27] and RocksDB [24]). An LSM-tree KV store comprises $n + 1$ levels on disk, denoted by L_0, L_1, \dots, L_n (from lowest to highest), and the capacity of a higher level L_i is a multiple (e.g., $10\times$ by default in LevelDB [27]) of that of a lower level L_{i-1} (where $1 \leq i \leq n$). It stores KV pairs in entirety as multiple disk files, called *SSTables*, in multiple levels. It also has two in-memory write buffers, namely *MemTable* and *Immutable MemTable*, and flushes the Immutable MemTable to level L_0 on disk with append-only writes. One main feature of the LSM-tree KV store is that all KV pairs in each of the levels from L_1 to L_n are *fully sorted* by keys to support fast scans, while KV pairs in L_0 are unsorted across different SSTables for fast flushes.

Write process. To write a KV pair, an LSM-tree KV store first inserts the KV pair into the MemTable, which keeps all buffered KV pairs sorted by keys in a skip-list. When the Memtable is full, it becomes an Immutable MemTable, which is then flushed to level L_0 as a new SSTable. The SSTable comprises the sorted KV pairs and some metadata (e.g., Bloom filters) for indexing. Meanwhile, the KV store generates a new MemTable to receive the subsequent new writes. When L_i is full (where $i \geq 0$), its SSTables will be integrated into L_{i+1} by a *compaction* operation. To compact an SSTable S from L_i into L_{i+1} , the KV store reads S and all SSTables in L_{i+1} that have overlapped key ranges with S , then sorts all KV pairs by keys and creates new SSTables. It finally writes them back into L_{i+1} . Thus, compaction induces to severe write amplification, which can reach up to a factor of $50\times$ [42].

Read process. To read a KV pair, an LSM-tree KV store first searches for the KV pair in memory. If the KV pair does not exist, the KV store performs binary search in each level of the LSM-tree, starting from the lowest level L_0 to the higher levels. For each level, it identifies an SSTable and checks its Bloom filter to see if the KV pair exists.

To scan KV pairs, the KV store first finds the starting key in each level. It then sequentially reads the KV pairs that fall in the queried range. It finally returns the set of KV pairs.

2.2 LSM-tree Optimizations and Limitations

We discuss two major classes of LSM-tree optimizations on how they reduce the compaction overhead of LSM-tree KV stores: (i) relaxing fully-sorted ordering [17, 52, 55] and (ii) KV separation [10, 20, 38, 42, 49, 51, 63]. Other types of optimizations are reviewed in §6.

Relaxing fully-sorted ordering. We consider PebblesDB [52], which realizes a *fragmented* LSM-tree, as a representative example, as shown in Figure 1(b). PebblesDB divides each level into several disjoint groups by *guards*, in which the key ranges of SSTables in the same group may overlap with each other. To compact a group of SSTables from L_i to

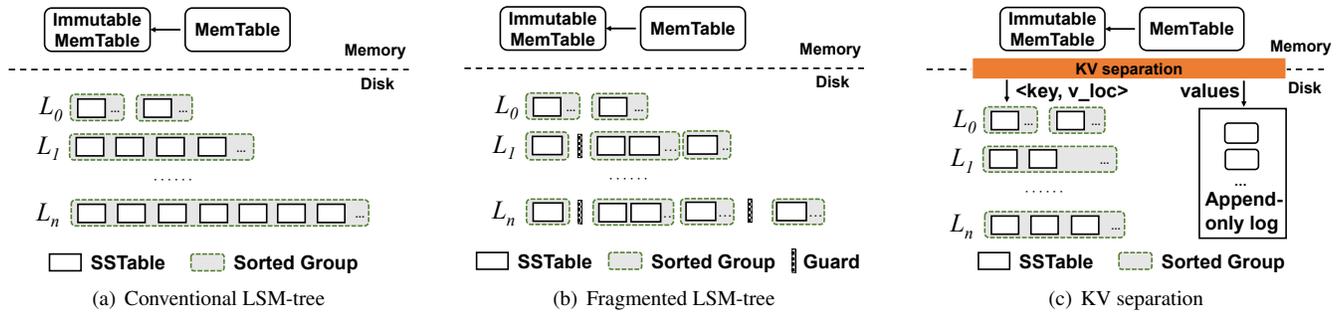


Figure 1: Overview of LSM-tree KV storage structure: (a) fully sorted within each level; (b) partially sorted within each level via multiple non-overlapping segments with guards; and (c) fully sorted for keys, unsorted for values.

L_{i+1} , PebblesDB only reads the SSTables within the group in L_i , sorts them to create new SSTables, and finally writes the new SSTables into L_{i+1} . In this case, a compaction operation in PebblesDB does not need to read SSTables from L_{i+1} , thereby greatly alleviating compaction overhead and write amplification. However, because of the overlapped key ranges of SSTables within each group, PebblesDB sacrifices scan performance. Although it can leverage multi-threading to issue reads in parallel, it incurs more CPU resources, while the improvement remains limited.

KV separation. Figure 1(c) illustrates the typical structure of KV separation in Wiskey [42] and Titan [51]. KV separation stores keys and values separately, in which keys and their references to values are treated as new KV pairs and stored in the LSM-tree, while the original values are separately stored in an append-only log, which is implemented as multiple blob files in Titan. For medium-to-large value sizes, as the keys and value references often have much smaller sizes than the original values. In this case, the total data volume in the LSM-tree is significantly reduced, so the compaction overhead and hence the write amplification are alleviated. In addition, a smaller LSM-tree reduces read amplification and hence improves read performance.

KV separation is a well-known optimization technique for LSM-tree KV stores, as KV pairs with large values sizes are commonly found in real-world KV workloads. For example, TiDB [30], a transactional database built atop a KV storage layer, maps every row of a database table into a KV pair whose value size can grow to hundreds of kilobytes. Atlas [38] maintains KV pairs of cloud storage with value sizes larger than 128 KB. Also, field studies [5, 44, 59] indicate that large-size values contribute to a considerable fraction of traffic in practical KV workloads, and the recent study by Facebook [9] shows that KV pairs for social graph data can have an average value size as large as 1 KB. Finally, the values with medium-to-large sizes (e.g., from 1 KB to 16 KB) are often chosen in the evaluation benchmarks of KV stores (e.g., [10, 42, 52]).

However, since KV separation writes values into an append-only log, the values of a consecutive range of keys are now scattered in different positions in the log. Thus, the scan oper-

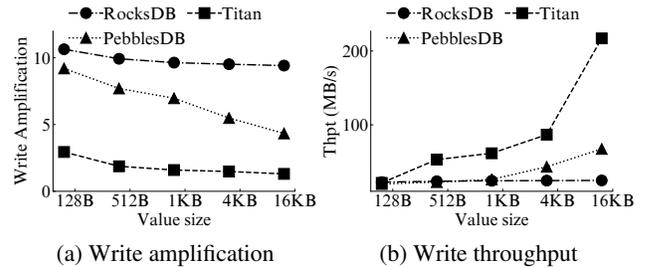


Figure 2: Write performance of RocksDB, PebblesDB, and Titan.

ations need to issue random reads to values, thereby leading to low scan performance in KV separation especially for the KV workloads whose values have small-to-medium sizes [5, 9] (e.g., in OLTP applications [5], more than 90% of values are smaller than 1 KB). Furthermore, KV separation needs to perform GC to reclaim the space of invalid values in the append-only log, and frequent GC operations incur extra I/O overhead [10].

2.3 Trade-off Analysis

We evaluate the design trade-offs of existing LSM-tree designs and optimizations. We consider three open-source KV stores: (i) RocksDB [24], which represents the state-of-the-art LSM-tree KV store with optimized performance; (ii) PebblesDB [52], which relaxes the fully-sorted ordering of each LSM-tree level for reduced compaction overhead; and (iii) Titan [51], which realizes KV separation with optimized implementation (e.g., managing values in multiple small Blob files instead of a large append-only log and using multi-threading to reduce GC overhead). We evaluate write amplification (i.e., the ratio of total write size to user write size) and performance, based on the testbed and configurations in §5.1.

Write performance. Figure 2 shows the write performance of loading a 100 GB database with different value sizes (varying from 128 bytes to 16 KB). From Figure 2(a), both PebblesDB and Titan significantly reduce the write amplification of RocksDB, and their write amplification ratios decrease as the value size increases. For example, for a value size of 16 KB, the write amplification ratios of RocksDB, PebblesDB, and Titan are $9.4\times$, $4.3\times$, and $1.3\times$, respectively.

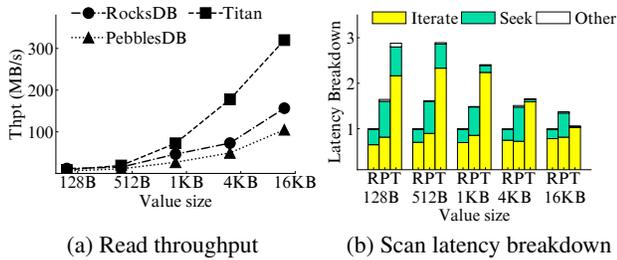


Figure 3: Read and scan performance of RocksDB (‘R’), PebblesDB (‘P’), and Titan (‘T’).

From Figure 2(b), both PebblesDB and Titan show higher write throughput than RocksDB due to the reduced write amplification. For example, for a value size of 16 KB, the write throughput gains of PebblesDB and Titan over RocksDB are $2.7\times$ and $8.7\times$, respectively. In short, the LSM-tree optimizations of relaxing fully-sorted ordering and KV separation are effective to reduce write amplification and hence increase write throughput, especially for large-size values.

Read and scan performance. Figure 3 shows the read and scan performance of RocksDB, PebblesDB, and Titan. For read performance, we issue read requests (i.e., random point queries) to a randomly loaded 100 GB KV store; for scan performance, we issue scan requests to 100 KV pairs. From Figure 3(a), relaxing fully-sorted ordering degrades the read performance, so the read throughput of PebblesDB is lower than that of RocksDB. Titan uses KV separation, which largely reduces the LSM-tree size, so its read throughput is evidently higher than that of RocksDB; for example, its throughput $2.0\times$ of that of RocksDB for a value size of 16 KB.

For scans, both PebblesDB and Titan perform worse than RocksDB, especially for small-to-medium size values. Figure 3(b) provides a latency breakdown for scans. For example, for a value size of 1 KB, the scan latencies of PebblesDB and Titan are $1.5\times$ and $2.4\times$ of that of RocksDB, respectively. Most of the scan time is spent on iteratively reading values (e.g., more than 90% for Titan). As the value size becomes larger (e.g., 16 KB), the differences of scan latencies among the KV stores are smaller, as accessing large-size values has smaller random read overhead. For the KV workloads that are dominated by small-size values [5, 9], the scan performance of PebblesDB and Titan will be limited in practice.

In short, existing LSM-tree designs and optimizations are subject to the performance trade-offs between reads/writes and scans. While relaxing the degree of ordering for values (e.g., using the fragmented LSM-tree or an unsorted append-only log in KV separation) reduces write amplification and improves write throughput, it sacrifices the scan performance, especially for values with small-to-medium sizes.

3 DiffKV Design

We present DiffKV, a novel LSM-tree KV store that aims for balanced performance in writes, reads, and scans.

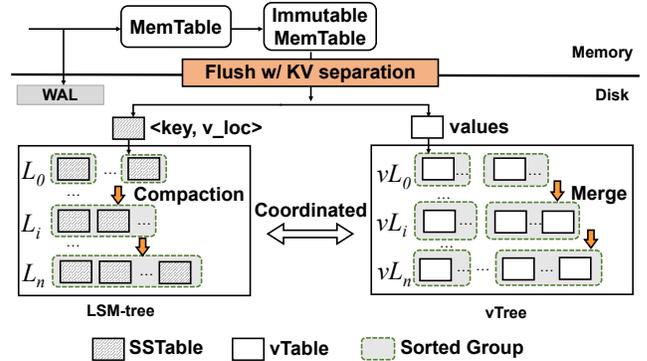


Figure 4: System overview.

3.1 System Overview

Figure 4 depicts the system architecture of DiffKV. DiffKV builds on KV separation for on-disk KV pairs, such that keys and values are decoupled during the flush of KV pairs from memory to disk and then they are separately stored. It also realizes *partially-sorted ordering* for values, so as to maintain high scan performance. To achieve this, DiffKV leverages a new LSM-tree-like multi-level tree, called the *vTree*, to manage values. As in the LSM-tree, the *vTree* comprises multiple levels, each of which can only be written in an append-only way. The difference between the *vTree* and the LSM-tree is that the *vTree* only stores values which are not necessarily fully sorted by their keys within each level; instead, they are allowed to be partially sorted for high scan performance.

To realize partially-sorted ordering for values, the *vTree* also necessitates a compaction-like operation as in the LSM-tree, which we call a *merge* operation to differentiate itself from a compaction operation in the LSM-tree. To reduce the merge overhead, DiffKV makes the compaction of the LSM-tree and the merge of the *vTree* be executed in a *coordinated* manner so as to reduce the overall overhead.

To make DiffKV compatible with existing LSM-tree store designs, it still follows the same in-memory data management as in conventional LSM-tree KV stores with an on-disk write-ahead log (WAL). Specifically, as depicted in Figure 4, DiffKV first writes KV pairs to the WAL, inserts them into the MemTable in memory, and finally flushes the Immutable MemTable to disk with KV separation.

3.2 Data Organization

The *vTree* adopts hierarchical data organization. It consists of multiple levels. Each level consists of *sorted groups*, and each sorted group further consists of multiple *vTables* (Figure 4). In the following, we elaborate their design details.

vTable. DiffKV organizes values as *vTables*, each of which has a fixed size (e.g., 8 MB by default). Note that each flush of an immutable MemTable may generate multiple *vTables* depending on the value size and MemTable size.

A *vTable* includes a *data area* that stores the values of KV pairs in a sorted order based on their keys, as well as

a *metadata area* that records the necessary metadata, such as the data size of the vTable, and the smallest and largest keys of the values in this vTable. Note that the Bloom filter is not required in the vTable (as opposed to the SSTable in the LSM-tree), since the values are still indexed by their keys in the LSM-tree. Thus, the metadata in each vTable has a very small size and brings limited storage overhead.

Sorted group. Each sorted group is a collection of vTables, and all vTables in a sorted group are fully sorted according to their corresponding keys. In other words, the key ranges of any two vTables in a sorted group have no overlaps. For ease of presentation, we also apply the concept of a sorted group for the LSM-tree, such that any set of SSTables in the LSM-tree that are fully sorted are also called a sorted group (e.g., all SSTables in the same level in the LSM-tree form a sorted group). In DiffKV, all vTables generated in one flush form a sorted group, so as to preserve the ordering of values in each immutable MemTable. We use the number of sorted groups as an indicator to measure the degree of ordering in the vTree. As the number of sorted groups increases, the degree of ordering decreases. In one extreme, if all SSTables/vTables form one sorted group, then we have the maximum degree of ordering, as all KV pairs are fully sorted.

vTree. The vTree consists of multiple levels, each of which is formed by multiple sorted groups. While the values in each sorted group are fully sorted, the values in a level of the vTree are not necessarily fully sorted, as they may belong to multiple sorted groups that have overlaps in the key range. As the vTree allows each level to have multiple sorted groups, a merge operation does not need to sort all values in successive two levels in the vTree; this alleviates the I/O overhead as opposed to the compaction in the LSM-tree.

3.3 Compaction-Triggered Merge

The vTree regularly performs merge operations to have partially-sorted ordering for values. Each merge reads a number of vTables and checks which values in the vTables remain valid. This can be done by querying the LSM-tree to retrieve the latest value location. Also, each merge needs to update the LSM-tree for the latest value locations of the valid values. To limit the merge overhead in the vTree, the merge operations in the vTree are not executed independently, but are triggered by the compaction operations in the LSM-tree in a coordinated manner. We call such a merge operation to be a *compaction-triggered merge* operation.

To explain the idea, we consider a simplified case in which we let each level in the vTree be coupled with only one level in the LSM-tree, and use L_i and vL_i to denote level i of the LSM-tree and vTree, respectively. If L_i and vL_i are correlated, then it means that for each KV pair, if the key is stored at level L_i in the LSM-tree, then the value of the KV pair can only be stored at level vL_i in the vTree. However, level vL_i in the vTree is not required to preserve the same fully-sorted ordering with its correlated level L_i in the LSM-tree.

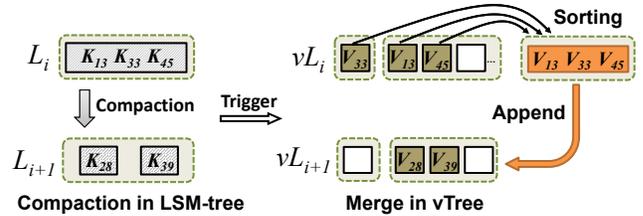


Figure 5: Compaction-triggered merge.

Figure 5 shows the idea of a compaction-triggered merge, which operates as follows. When a compaction operation is triggered to compact keys from L_i to L_{i+1} in the LSM-tree, it also triggers a merge operation that moves the corresponding values from vL_i to vL_{i+1} in the vTree. In the merge operations, there are two major issues: (i) which values should be involved in the merge and (ii) how to write back these values to level vL_{i+1} in the vTree. First, we merge only the values at level vL_i , and their corresponding keys must participate in the compaction in the LSM-tree. We call the value whose key participates in the compaction to be the *compaction-related value* for ease of presentation. Second, we reorganize all compaction-related values at level vL_i to generate new vTables and write these vTables to level vL_{i+1} in an append-only manner. For the example in Figure 5, the values $V_{33}, V_{13}, V_{45}, V_{28}, V_{39}$ are compaction-related values, and V_{33}, V_{13}, V_{45} will be involved in the merge, and finally appended to level vL_{i+1} after sorting.

Note that all values in the generated vTables in each merge are fully sorted; that is, they form only one single sorted group. However, we point out that the merge operation does not require to reorganize all vTables in both levels of vL_i and vL_{i+1} in the vTree. That is, when merging, a new sorted group is created in level vL_{i+1} with all vTables from level vL_i it. This avoids the rewrites of all values at level vL_{i+1} , thereby mitigating write amplification. In addition, the old vTables at vL_i will not be deleted during a merge, as they may still contain valid values, and they will be reclaimed later by GC.

The benefits of a compaction-triggered merge are two folds. First, merging only compaction-related values makes it very efficient to identify which values are still valid, as the corresponding keys also need to be read out from the LSM-tree during a compaction. In contrast, if the vTree is independent of the LSM-tree and triggers merge operations independently, then it needs to query the LSM-tree and compare the value locations to determine the validity of values, thereby inevitably incurring large query overhead. Second, as the locations of valid values are changed when generating new vTables during a merge operation, the LSM-tree needs to be updated accordingly to maintain the latest value locations. Since only the compaction-related values are merged, updating the value locations in the LSM-tree can be executed by directly updating the KV pairs participating in the compaction. Thus, the overhead of updating value locations can be hidden in the compaction operation as the compaction itself needs to rewrite the KV pairs in the LSM-tree.

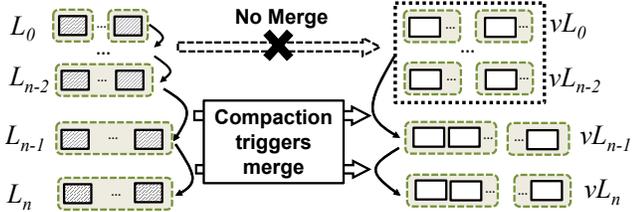


Figure 6: Lazy merge.

3.4 Merge Optimizations

Compaction-triggered merge operations incur limited merge overhead caused by checking the validity of values and writing back new value locations into the LSM-tree. However, letting each compaction operation trigger a merge operation may cause frequent merge operations. For example, if each level in the v Tree is correlated with only one level in the LSM-tree, then each compaction operation must trigger a merge operation in the v Tree. To further reduce the merge overhead in the v Tree, we propose two merge optimizations.

Lazy merge. We propose *lazy merge* to limit the merge frequency, and hence the merge overhead, in DiffKV. Our idea is to aggregate multiple lower levels in the v Tree as a single level, and correlate the aggregate level with multiple levels in the LSM-tree. Specifically, as depicted in Figure 6, we aggregate all levels vL_0, \dots, vL_{n-2} in the v Tree as a single level and correlate the aggregate level with levels from L_0 to L_{n-2} in the LSM-tree. Thus, any compaction between level L_0, \dots, L_{n-2} will not trigger a merge operation; in other words, the merge operations between level vL_0, \dots, vL_{n-2} will be delayed unless the values need to be merged into level vL_{n-1} .

Lazy merge significantly reduces the number of merge operations and the amount of data size being merged, but sacrifices the degree of ordering for the values in lower levels in the v Tree. Nevertheless, we argue that the sacrifice poses limited degradation to the scan performance. Recall that the LSM tree increases its capacity toward higher levels (e.g., the size of L_i is $10\times$ of that of L_{i-1} in LevelDB [27]) (§2.1). Thus, the last two levels L_{n-1} and L_n contain the majority of KV pairs. The uneven data distribution across levels implies that most values are actually retrieved from the last two levels of the v Tree for scans, so the degree of ordering of values in the last two levels is the dominant factor that determines the scan performance. In other words, the low levels of the v Tree only have limited impact on scan performance, and the frequent merge operations in the low levels do not help scan performance but instead incur large merge overhead.

Scan-optimized merge. We adjust the degree of ordering for values in the v Tree via scan-optimized merge, so as to maintain high scan performance. Recall that in a compaction-triggered merge operation, say merging the values from vL_i to vL_{i+1} , only the values in the lower level vL_i are reorganized and appended to the higher level vL_{i+1} , while the values in level vL_{i+1} are not involved in the merge operation and will

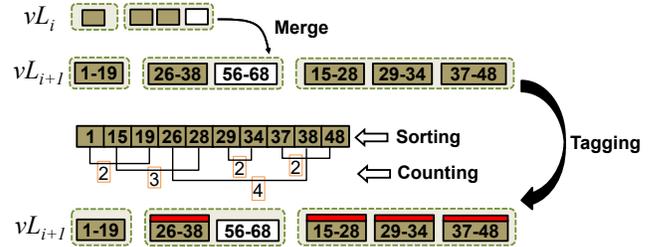


Figure 7: Scan-optimized merge.

not be sorted (§3.3). This append-only merge policy mitigates write amplification, but may result in too many sorted groups, which may have overlapped key ranges as values are not sorted across sorted groups. Thus, our idea is to find out the v Tables which have overlapped key ranges with many other v Tables, and also make them participate in the merge process regardless of which level they reside. With this, we can preserve a higher degree of ordering for values in the v Tree, and thus benefit the scan performance.

Figure 7 depicts the idea of scan-optimized merge. After the normal compaction-triggered merge, we further check the v Tables that contain compaction-related values at level vL_{i+1} . Our goal is to identify the set of v Tables that satisfy two conditions: (i) at least one v Table in the set has overlapped key range with others, and (ii) the number of v Tables (i.e., the set size) is larger than a pre-defined threshold, denoted by `max_sorted_run`. The rationale is that if such a set of v Tables exists, then the scan performance may degrade as these v Tables are not in a sorted order. We add a *scan optimization tag* for these v Tables, so that they will always participate in the next compaction-triggered merge and increase the degree of ordering for values in tagged v Tables.

To identify the set of v Tables for tagging, we first retrieve the start and end keys of each v Table containing compaction-related values at level vL_{i+1} , and sort these keys. For each checked v Table, we count the number of v Tables that have overlapped key ranges with it, and this can be done by scanning once the sorted key string. For example, as in Figure 7, consider a checked v Table [26-38]. By scanning the sorted string, we can count the number of start keys before key 38 (i.e., five in this case) and the number of end keys before key 26 (i.e., one in this case). By subtracting the two numbers, we can obtain the number of v Tables that have overlapped key ranges with v Table [26-38], which is four including itself (i.e., v Tables [15-28], [29-34], [37-48], and [26-38]). Finally, if the number of v Tables with overlapped key ranges is larger than the threshold `max_sorted_run`, then we add a scan optimization tag for all these v Tables to include them in the next compaction-triggered merge. We persist the optimization tags in a *manifest file*, which is already used by existing systems to track the version changes of KV pairs after each compaction; the persistence overhead is negligible.

Scan-optimized merge is an enhancement to compaction-triggered merge: a compaction-triggered merge operation only

appends values in a lower level to its next higher level, while a scan-optimized merge operation further includes certain values in the higher level into the merge to increase the degree of ordering of values in the vTree. Note that scan-optimized merge incurs limited merge overhead (see Figure 15 in §5.4) for two reasons. First, we allow each level in the vTree to have multiple sorted groups (i.e., the whole level is not necessarily fully sorted). Second, not all values in a tagged vTable participate in the merge, but instead only the compaction-related values are merged.

3.5 Garbage Collection

The vTree rewrites compaction-related values to new vTables (§3.3), so it necessitates garbage collection (GC) to reclaim the space of invalid values (in the LSM-tree, its invalid data is reclaimed via compaction). To reduce the GC overhead, we propose a *state-aware lazy approach* based on the amount of invalid values in each vTable.

State awareness. DiffKV tracks the amount of invalid values in each vTable in a hash table. Each time when a vTable participates in a merge operation, DiffKV counts the amount of values being retrieved from the vTable and updates the amount of invalid values in the old vTable in the hash table. It also inserts an entry for any new vTable in the hash table. The updates to the hash table are executed during a merge operation, so the overhead is limited. Also, each entry in the hash table only occupies few bytes for each vTable, so the memory overhead of the hash table is limited.

Lazy GC. DiffKV takes a lazy approach to limit the GC overhead. It selects a vTable as a GC candidate if the vTable has a fraction of invalid values greater than a predefined threshold (denoted by `gc.threshold`). Note that DiffKV does not immediately reclaim the candidate vTables; instead, it simply marks a *GC tag* for each candidate, and delays the GC until the next compaction-triggered merge. Specifically, if a vTable with a GC tag is involved in a compaction-triggered merge, the values contained in this vTable will always be rewritten to the next higher level (similar to scan-optimized merge).

Lazy GC avoids the extra overhead of querying the LSM-tree for the validity of values in the candidate vTable and updating the LSM-tree for the new locations of the valid values. It is now delayed to be executed together with the merge operation, so that the overhead of querying and updating the LSM-tree can be hidden within the merge operation.

3.6 Discussion

Optimizing compaction at L_0 . As KV separation is executed during flushes, SSTables at level L_0 in the LSM-tree may be very small, especially when KV pairs are large, so we propose a simple optimization called *selective compaction* to aggregate small SSTables at L_0 . Specifically, we trigger intra-level compaction which simply combines multiple small SSTables at L_0 to generate a new large one without merging with SSTables at L_1 . With selective compaction, the size of

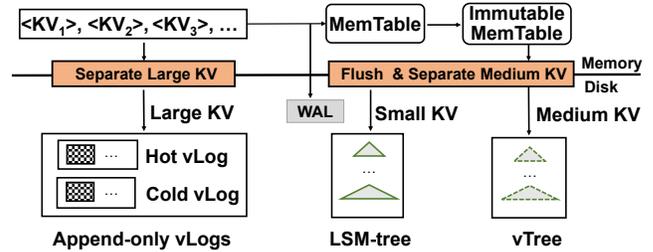


Figure 8: Fine-grained KV separation.

SSTables at L_0 will eventually become comparable to that of SSTables at L_1 , and hence no extra compaction overhead will be introduced due to KV separation.

Crash consistency. DiffKV is implemented on Titan (which builds on RocksDB) (§5), and provides the same level of consistency as RocksDB after system crashes. To guarantee data consistency, DiffKV uses a write-ahead log (WAL), and writes KV pairs to the WAL before writing to the MemTable. Also, DiffKV provides crash consistency for the hash table that records the amount of invalid data in each vTable (§3.5). As the hash table is updated after compaction, DiffKV appends the update information into the manifest file (§3.4) after compaction, so it can be restored when DiffKV recovers from a crash.

4 Fine-grained KV Separation

The benefit of KV separation is significant for large KV pairs, but diminishes for small KV pairs (§2.3). However, mixed workloads with varying value sizes are also common; for example, the value size may vary in a large range under the generalized Pareto distribution [25]. In this section, we further enhance DiffKV via fine-grained KV separation by distinguishing KV pairs by value sizes, so as to achieve balanced performance under mixed workloads.

4.1 Differentiated Value Management

DiffKV classifies KV pairs into three groups based on the value size, by using two parameters, namely `value_small` and `value_large`, as depicted in Figure 8. For KV pairs whose value size is larger than `value_large` (i.e., large KV pairs), DiffKV adopts KV separation, and manages values with a hotness-aware multi-log design called *vLogs*. For KV pairs whose value size is between `value_small` and `value_large` (i.e., medium KV pairs), DiffKV stores values in the vTree and keep both keys and value locations in the LSM-tree. For KV pairs whose value size is smaller than `value_small` (i.e., small KV pairs), DiffKV bypasses KV separation and stores the whole KV pairs directly in the LSM-tree. As a result, DiffKV achieves balanced performance for different value sizes.

Note that for large KV pairs, DiffKV adopts the workflow as in Wiskey [42], in which KV separation is performed before writing to MemTable (see Figure 8). Specifically, DiffKV directly flushes the values of large KV pairs into vLogs, treats their keys and value locations as new KV pairs, and writes

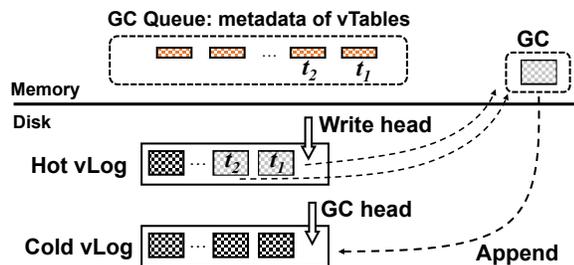


Figure 9: GC for vLogs.

them to MemTable as regular user data. The benefits of performing KV separation early for large KV pairs are two-fold. First, by directly flushing large-size values into vLogs and keeping only small-size value locations in the MemTable, we can save substantial memory space, while still guaranteeing high write performance due to large sequential I/Os. Second, as large-size values are first written to disk, there is no need to write them to the WAL, and this reduces the amount of I/Os. Note that for small and medium KV pairs, as well as the keys and value locations for large KV pairs, they still need to be written to the WAL so as to guarantee consistency (§3.6).

4.2 Hotness-aware vLogs

Structure of vLogs. A vLog is designed as a simple circular append-only log, which consists of a set of *unsorted vTables*. Unsorted vTables share a similar storage format with vTables described in §3.2, and the only difference is that values are written to *unsorted vTables* with append, so they are not sorted even within each unsorted vTable. The reason why we do this is because KV separation for large KV pairs is performed before writing to the MemTable, and values for large KV pairs are flushed to disk immediately after KV separation so as to avoid writing to WAL, so there is no way to sort the values in each vTable (see Figure 8). In fact, there is no need to sort these values as they have a large size, and hence they can already benefit from large I/Os without batched writes.

GC for vLogs. To reduce GC overhead, we leverage a hotness-aware design by employing a simple yet efficient parameter-less hot-cold separation scheme. As shown in Figure 9, we adopt two vLogs, namely a *hot vLog* and a *cold vLog*, to store hot and cold values, respectively. Each vLog has its own write frontier, and we call them *write head* and *GC head*, respectively. To realize hot-cold separation, the data from user writes are appended to the write head in the hot vLog, and the data from GC writes (i.e., the valid values that need to be written back after GC) are appended to the GC head in the cold vLog. The rationale is that the values reclaimed by GC are usually accessed less frequently than the recently written user data, so they can be regarded as cold data. One benefit of this design is that it is simple to implement, as no parameter is required to realize hot-cold identification. Clearly, we can also apply alternative hotness-aware classification schemes.

DiffKV employs a *greedy* algorithm to reduce GC cost, and the idea is to reclaim the unsorted vTables which have

the largest amount of invalid values. Specifically, DiffKV monitors the ratio of invalid values of each unsorted vTable during compaction, and maintains a GC queue in memory to track all candidate vTables, which are the unsorted vTables with the ratio of invalid values being greater than the threshold `gc_threshold`. Note that the GC queue only keeps the metadata of each unsorted vTable, and it is maintained in a descending order according to the ratio of invalid values. When GC is triggered, DiffKV simply selects unsorted vTables tracked in the queue head (e.g., t_1 , and t_2 in Figure 9), then appends the valid values in the selected vTables to the GC head in the cold vLog. For performance consideration, DiffKV implements GC as a background process with multiple GC threads.

5 Evaluation

In this section, we evaluate and compare DiffKV with the three state-of-the-art KV stores introduced in §2.3: RocksDB [24], PebblesDB [52], and Titan [51]. For fair comparisons with Titan, we also implement DiffKV based on it and our modifications contain around 2.1K lines of code.

5.1 Setup

Testbed. We conduct experiments on a single machine equipped with an 12-cores Intel Xeon E5-2650v4 CPU, 16 GB memory, and Samsung 860 EVO 480 GB SSD. The machine runs Ubuntu 18.04 LTS with Linux kernel 4.15.

Workload. We modify YCSB-C [48, 54], a C++ version of YCSB [14], to generate workloads based on the workload statistics in [9]. We fix the key size as 24 bytes, and configure the value size with the generalized Pareto distribution [29], whose probability density function is:

$$f(x) = (1/\sigma)(1 + k(x - \theta)/\sigma)^{-1-1/k} \quad (1)$$

where x represents the value size, k , θ , and σ are adjustable parameters. By default, we limit the maximum value size as 128 KB, and set $k = 0.92$, $\sigma = 226$, $\theta = 0$ by using the most common workload setting in [25]. Under this setting, the average value size is 1 KB.

System configuration. We refer to the official tuning manual for experimental verification [26]. For all KV stores, we use the recommended configuration by setting MemTable size as 64 MB, SSTable size as 16 MB, configure Bloom filters by setting 10 bits/key. Since our testbed machine has 12 cores, to speedup compactions and also limit overall CPU usage, we use 8 background threads for compaction by following the optimization tuning guide. We also allocate 8 GB memory for block cache and leave the rest for page cache in operating system. For Titan, as GC affects both space usage and foreground write performance, we consider three cases: (i) *no GC (No-GC)*, which achieves the best write performance, but incurs the largest space overhead; (ii) *background GC (BG-GC)*, in which GC is executed in the background without blocking foreground writes; and (iii) *foreground GC (FG-GC)*, in which a limit on space usage is set and GC may block

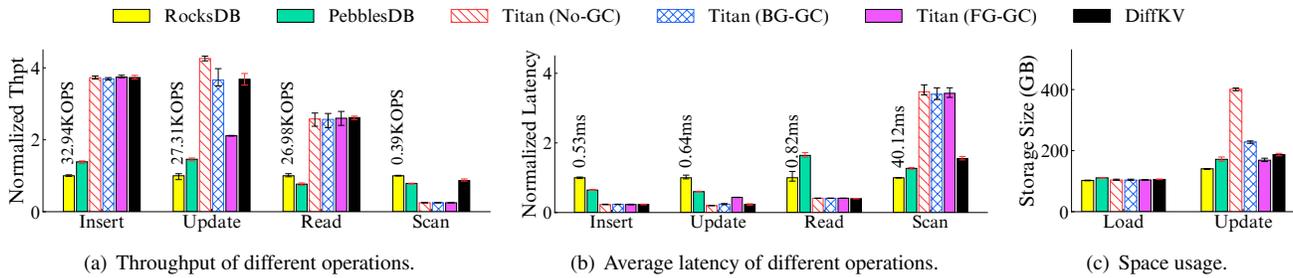


Figure 10: Microbenchmarks on RocksDB, PebblesDB, Titan, and DiffKV.

foreground writes if the free space drops below the predefined limit. For DiffKV, the two thresholds for differentiating small, medium, and large KV pairs are set as 128 bytes and 8 KB, respectively. We limit the vTable size as 8 MB, and set a GC tag to a vTable if it contains more than 30% invalid data. We study the impact of these parameters in §5.6. Each experiment was run at least five times.

5.2 Microbenchmarks

We first study the throughput and latency performance of various KV operations with the following workloads: (i) insert 10 GB KV pairs, (ii) update 300 GB KV pairs, (iii) read 10 GB KV pairs, and (iv) scan 10 GB KV pairs. We first randomly load 100 GB KV pairs, then issue the requests of each workload, and finally clear all KV pairs from the KV store to avoid interference. For scans, we use the widely used configuration for performance evaluation [8, 14, 43, 65], i.e., we issue 16 scan threads, each of which reads 100 KV pairs. We also study the impact of other scan settings in §5.5. By default, we use a Zipf distribution with the skewness parameter 0.9 for each workload. We also run our evaluation under uniform workloads and observe similar conclusions, so we omit the results in the interest of space.

Throughput. Figure 10(a) shows the throughput results, normalized with respect to the throughput of RocksDB for ease of comparison. Compared to RocksDB and PebblesDB, DiffKV achieves $3.8\times$ and $2.7\times$ insert throughput, $3.7\times$ and $2.3\times$ update throughput, $2.6\times$ and $3.4\times$ read throughput, respectively. Thus, DiffKV significantly improves both write and read throughput, with comparable scan performance.

Compared to Titan, DiffKV always improves the scan performance significantly, regardless of the GC policy used in Titan. For example, DiffKV achieves $3.2\times$ scan throughput over Titan. For write performance, even compared to the case of NO-GC (i.e., the case of best write performance for Titan), DiffKV still has similar write performance. Furthermore, compared to the case of using foreground GC in Titan, DiffKV achieves $1.7\times$ update throughput. Note that since the throughput results are evaluated in OPS, the throughput of scans is much lower than that of other operations.

Average latency. Figure 10(b) shows the latency results. Compared to RocksDB and PebblesDB, DiffKV reduces the latency of inserts, updates, and reads by up to 63.8%-78.1%,

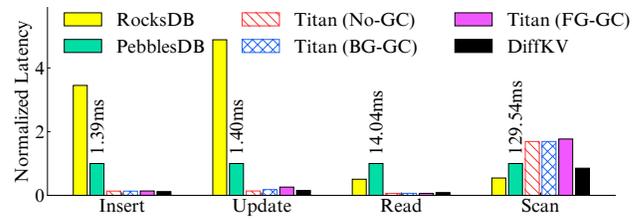


Figure 11: Tail latency of different operations.

Workload	Statistics
A (Update Heavy)	50% updates, 50% reads
B (Read Mostly)	5% updates, 95% reads
C (Read Only)	100% reads
D (Read Latest)	5% inserts, 95% reads
E (Scan Mostly)	5% inserts, 95% scans
F (Read-Modify-Write)	50% read-modify-write, 50% reads

Table 1: YCSB core workloads.

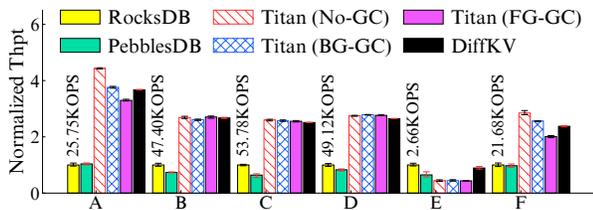
with a similar scan latency. Compared to Titan, DiffKV has similar latencies for inserts, writes, and reads, while reducing the scan latency by up to 43.2%.

Space usage. Figure 10(c) shows the space usage under GC, in which we randomly load 100 GB KV pairs and update 300 GB KV pairs with a Zipf distribution. As no GC is triggered in the load phase, all KV stores show the same space usage. However, after the update phase, Titan incurs large space usage if it disables GC or uses background GC only. DiffKV reduces space usage by up to 18%-53.7% compared to Titan (NO-GC) and Titan (BG-GC).

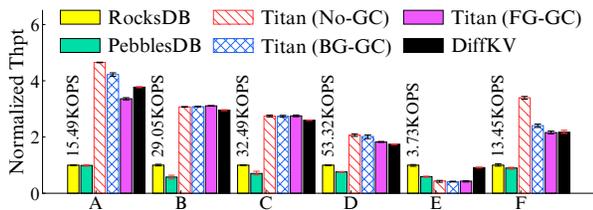
Tail Latency. We evaluate the 99-th percentile tail latency as shown in Figure 11. We normalize the results with respect to PebblesDB. Compared to RocksDB and PebblesDB, DiffKV has a much lower tail latency for inserts, updates, and reads. For example, it reduces the tail latency of inserts, updates, and reads of RocksDB by 96.5%, 94.3%, and 82.7%, respectively, while keeping similar tail latency for scans. Compared to Titan, DiffKV has a similar tail latency for inserts, updates and reads, but reduces the scan tail latency by 50.4%.

5.3 YCSB Evaluation

We show the performance of DiffKV under the YCSB workloads (Table 1). Each workload performs 100M operations on a randomly loaded 100 GB data store, other settings are the same as before. We consider both uniform distribution



(a) Zipfian workload.



(b) Uniform workload.

Figure 12: Performance under YCSB workloads.

and Zipf distribution with parameter 0.9, except workload D which reads the latest data as defined by the benchmark [14].

Throughput. Figure 12 shows the throughput results, normalized with respect to the throughput of RocksDB. Compared to RocksDB, DiffKV achieves 1.7-4.5 \times throughput for all read- or write-intensive workloads (i.e., all except workload E), and achieves similar performance under the scan-dominant workload E. Note that RocksDB has KV pairs fully sorted in each level, so it represents the best case in terms of scan performance. Compared to Titans, DiffKV achieves 2 \times scan throughput, while keeping similar performance under other workloads regardless of whether background GC or foreground GC are used. In short, DiffKV achieves balanced performance in all aspects.

Tail latency. Figure 13 shows the tail latency results. As YCSB workloads are mixed with different operations with highly different latencies, we show the tail latency of each type of operations. We normalize the results with respect to Titan (NO-GC). We observe a similar conclusion as in the case of microbenchmarks. That is, compared to RocksDB and PebblesDB, DiffKV significantly reduces the tail latency of inserts, updates and reads; compared to Titan, DiffKV reduces the tail latency of scans, so it always performs almost the best in all performance aspects.

Space usage. We show the space usage under YCSB workload A, which has 50% updates. We observe similar results as in Figure 10(c), i.e., DiffKV increases the space usage by 11.9% and 0.7% compared to RocksDB and PebblesDB, respectively. Also, the LSM-tree, v Tree, and v Logs incur 5%, 63.5%, and 31.5% of space in DiffKV, respectively.

5.4 Analysis on Merge Optimizations

We evaluate the merge optimization techniques of DiffKV (§3.3-§3.4) and show how they address the write-scan trade-

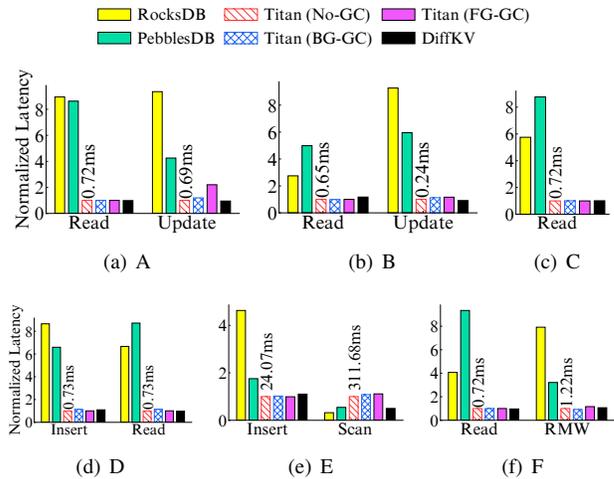


Figure 13: Tail latency of different ops under YCSB workloads.

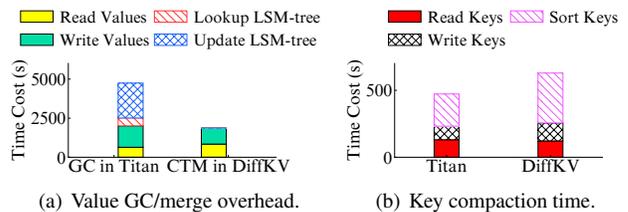


Figure 14: Impact of coordinated value management in DiffKV: it significantly reduces the value merge overhead, and slightly increases key compaction overhead.

off and realize the balanced performance for DiffKV.

We first compare the merge overhead in DiffKV with the GC overhead in Titan. Here, we deploy only the compaction-triggered-merge (CTM) in DiffKV and focus on the effectiveness of the coordinated design. Figure 14(a) shows the time cost breakdown for GC in Titan and merge in DiffKV. We issue a workload of updating 300 GB KV pairs with a Zipf distribution on a randomly loaded 100 GB KV store, using the default background GC for Titan. DiffKV costs much less time than Titan for value management, with a 60.7% reduction of time cost. The time saving mainly comes from the coordinated merge design, in which the overhead of looking up the LSM-tree and updating the new value locations to LSM-tree can be avoided via the compaction-triggered merge. However, as compaction must wait for the completion of the triggered merge, the compaction time slightly increases. Such an overhead can be mitigated via the merge optimizations (i.e., lazy merge and scan-optimized merge).

Figure 15 studies the impact of the merge optimizations. Lazy merge (LM) further reduces the number of merge operations and the amount of merged data size compared to using only the compaction-triggered merge (CTM), by 65.5% and 66.2%, respectively. However, it increases the number of sorted groups in v Tree, which is the key factor of influencing scan performance. For example, lazy merge adds up to 20% sorted groups in the last two levels. By further in-

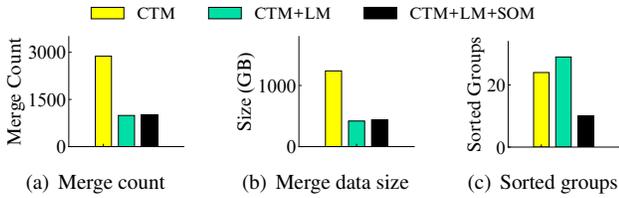


Figure 15: Impact of merge optimizations on merge overhead.

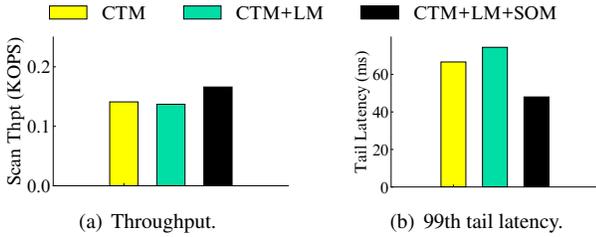


Figure 16: Scan performance.

corporating scan-optimized merge (SOM), the number of sorted groups reduces significantly, by up to 68% compared to lazy merge. Meanwhile, the merge overhead keeps almost the same. Due to the reduced number of sorted groups, the scan performance improves. For example, as shown in Figure 16, scan-optimized merge increases 18% of scan throughput and reduces 27% of the 99th tail latency compared to lazy merge. In summary, by combining lazy merge and scan-optimized merge with the coordinated design, DiffKV guarantees desired ordering for values with limited merge overhead, and hence achieves balanced performance in all aspects.

5.5 Scan Performance

Recall that the main benefit of DiffKV over Titan is its scan improvement. We further examine the scan performance by varying the *scan length* (i.e., the number of KV pairs read by each scan) and the number of scan threads (i.e., the number of threads initiated by clients for concurrent access).

Figure 17(a) shows the scan performance versus the scan length (from 20 to 10000), while fixing the total read size as 10 GB. As GC does not influence scan performance for Titan, we focus on only the case without GC. DiffKV significantly outperforms Titan, and as the scan length increases, the performance gain further increases. The reason is that Titan stores values in an unsorted manner and hence has poor scan performance, while DiffKV maintains a partial ordering for values. Even compared to RocksDB, which represents the optimum for scan, DiffKV still reaches 83% of scan throughput when the scan length is 20, and the ratio further increases to 96% when the scan length increases to 10000. Thus, DiffKV achieves similar scan performance with RocksDB.

Figure 17(b) studies the impact of the number of scan threads (from 8 to 64), while fixing the scan length as 100. DiffKV still significantly outperforms Titan, and achieves comparable performance with RocksDB under all settings.

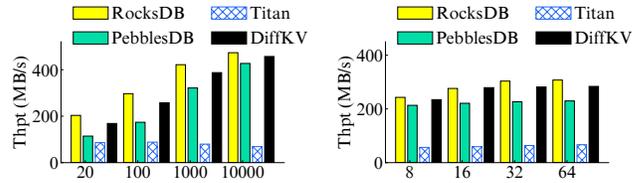


Figure 17: Scan performance under different settings.

Figure 17: Scan performance under different settings.

5.6 Tunable Parameters

We analyze the sensitivity of the parameters in DiffKV. First, the parameter `value_small` is used to differentiate small and medium KV pairs. In other words, for each value size, if KV separation brings no benefit to write, then we should treat this kind of KV pairs as small ones and keep them in the LSM-tree; otherwise, we should leverage KV separation and store values in `vTree`. Thus, to find an appropriate setting for parameter `value_small`, we compare the write performance of using the LSM-tree and the `vTree` under different value sizes. Figure 18(a) shows the load throughput versus the value size. When the value size is at least 128 bytes, using the `vTree` improves write performance (i.e., KV separation is beneficial). Thus, we set `value_small` as 128 bytes by default.

Second, we justify how to set an appropriate value for `value_large`, which influences both the write and scan performance. A smaller `value_large` means more KV pairs are regarded as large ones and stored in `vLogs`. We focus on a mixed workload generated with the default parameters (§5.1), and show the write and scan performance by varying `value_large` from 1 KB to 32 KB. As shown in Figure 18(b), when `value_large` increases from 8 KB to 32 KB, the marginal improvement of the scan performance is very limited. This implies that managing KV pairs whose value sizes are larger than or equal to 8 KB with `vLogs` only slightly degrades the scan performance. On the other hand, for write throughput, compared to the case of setting `value_large` as 1 KB, it already achieves around 80% of the throughput when the parameter is set as 8 KB. Thus, we set `value_large` as 8 KB by default.

Third, the parameter `max_sorted_run` controls the number of sorted groups in each of the last two levels in `vTree`, so it influences the scan performance and merge overhead, i.e., it determines the write-scan trade-off. Figure 18(c) shows the write-scan trade-off by varying `max_sorted_run` from 15 to one. Here, we run a workload which scans 10 GB data on a randomly loaded 100 GB data store and each scan requests 100 KV pairs. When there are more than 10 sorted groups in each level, both the write and scan performance change slowly, so we set `max_sorted_run` as 10 by default.

Finally, we study the impact of `gc_threshold` on write performance and space usage. Recall that a `vTable` will be set with a GC tag if its invalid data exceeds the threshold defined by `gc_threshold`, so a smaller `gc_threshold` means more frequent GC, and both the write performance and space usage should

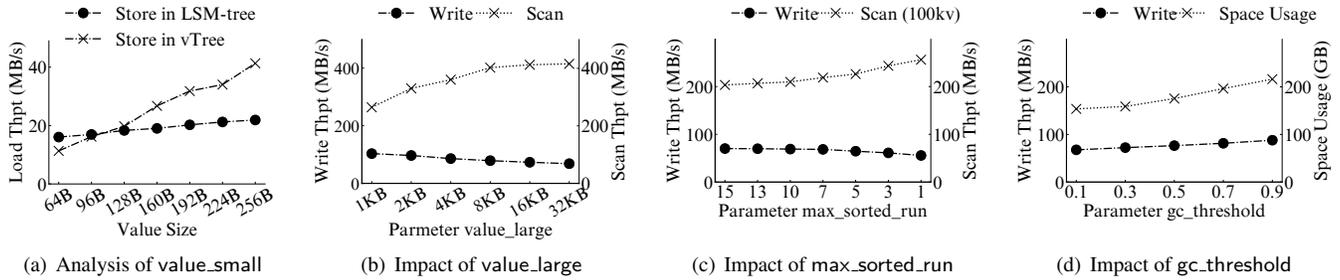


Figure 18: Empirical study on tunable parameters in DiffKV. Figure (a) justifies the setting of `value_small` by comparing the write performance of using LSM-tree and vTree under different value sizes. Figure (b) justifies the setting of `value_large` by studying its impact on write and scan performance. Figure (c) justifies the setting of `max_sorted_run`. Figure (d) shows the impact of `gc_threshold`.

decrease. Figure 18(d) shows the results. As `gc_threshold` increases, the increased write throughput becomes smaller. This implies that with the lazy GC, the impact of GC on write performance is limited. On the other hand, the space usage significantly increases (from 158.8 GB to 216.2 GB) when we increase `gc_threshold` from 0.3 to 0.9. Thus, we set `gc_threshold` as 0.3 by default.

6 Related Work

Research on KV stores has received a lot of attentions. In addition to building KV stores on new hardware like non-volatile memory [12, 13, 21, 34–36, 39, 44, 61] or characterizing real-world KV workloads [5, 9], many studies focus on optimizing the read/write performance of LSM-tree KV stores [6, 10, 15, 16, 32, 40, 42, 52, 56, 58, 62, 63].

Extensive efforts focus on reducing the compaction overhead of LSM-tree KV stores. One line of studies follows the idea of relaxing the fully-sorted ordering of KV pairs. PebblesDB [52] proposes a fragmented LSM-tree that relaxes the fully-sorted ordering of KV pairs by dividing each level into multiple non-overlapped segments and allowing KV pairs within each segment to be unsorted. dCompaction [47] delays compaction by constructing virtual SSTables that contain only metadata for multiple SSTables with overlapped ranges, and it can also be regarded as a relaxation of the fully-sorted ordering of KV pairs. Dostoevsky [17] introduces a lazy-leveling merge policy by adopting the leveling policy only for the last level and using tiering for other levels. The ideas of relaxing the fully-sorted ordering are also found in VT-Tree [57], LWC-tree [60], SlimDB [55], and SifrDB [45].

Another line of studies of mitigating the write amplification problem is KV separation. WiscKey [42] is the first work that proposes this technique by storing values in a separate append-only circular log, and Bourbon [15] extends WiscKey by integrating a learning approach for indexing the values. Based on KV separation, a lot of efforts are made to reduce the overhead of log management for values, especially the GC overhead [10, 20, 49, 51]. HashKV [10] leverages hash-based data grouping so as to reduce the GC cost of the circular log. BadgerDB [20] reuses the write-ahead log (WAL) as a value log, so as to save the data flush overhead. Titan [51] adopts KV

separation and leverage BLOB files for value management. We point out that all these studies follow the design of append-only logs for value management. In contrast, DiffKV proposes the vTree for maintaining the partially-sorted ordering for values, so as to realize balanced I/O performance.

Multiple studies also leverage hash indexing or optimize the Bloom filter to improve read and write performance. For hash indexing, LSM-trie [58] utilizes a hash-based trie structure to improve the performance for small KV pairs, while UniKV [63] unifies hash indexing and the LSM-tree to simultaneously improve both read and write performance. For the Bloom filter optimization, Monkey [16] proposes to differentiate Bloom filters between different levels; ElasticBF [40] further develops a fine-grained elastic Bloom filter scheme to improve read performance; SuRF [62] introduces a succinct range filter to optimize both reads and scans.

Unlike existing studies that usually possess performance trade-offs, DiffKV aims to realize balanced I/O performance by simultaneously improving the performance of writes, reads, and scans. DiffKV adopts KV separation, and takes one step further to adopt a new vTree structure with a coordinated design to realize the differentiated ordering for keys and values. It also adopts fine-grained KV separation, so as to realize balanced performance under mixed workloads.

7 Conclusion

In this paper, we propose to leverage differentiated ordering for keys and values to simultaneously achieve high performance for writes, reads, and scans. We develop DiffKV, which follows KV separation and utilizes a new structure vTree for value management with a partial ordering. By leveraging a coordinated design and multiple merge optimizations, DiffKV achieves efficient writes, reads, and scans with low storage cost, and thus realizes balanced performance in all aspects.

Acknowledgments: We thank our shepherd and the anonymous reviewers for their comments. This work is supported by National Key R&D Program of China (2018YFB1003204), NSFC (61772484, 61772486), Youth Innovation Promotion Association CAS, and USTC Research Funds of the Double First-Class Initiative (YD2150002003). Yongkun Li is USTC Tang Scholar. Yinlong Xu is the corresponding author.

References

- [1] 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 *Proc. of ACM SOSP*, 2019.
- [2] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng. Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O. In *Proc. of USENIX ATC*, 2017.
- [3] Apache. Cassandra. <http://cassandra.apache.org/>.
- [4] Apache. HBase. <https://hbase.apache.org/book.html>.
- [5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proc. of ACM SIGMETRICS*, 2012.
- [6] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proc. of USENIX ATC*, 2019.
- [7] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook’s Distributed Data Store for the Social Graph. In *Proc. of USENIX ATC*, 2013.
- [8] C. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan, Z. Liu, F. Zhu, and T. Zhang. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *Proc. of USENIX FAST*, 2020.
- [9] Z. Cao, S. Dong, S. Vemuri, and D. H. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-value Workloads at Facebook. In *Proc. of USENIX FAST*, 2020.
- [10] H. Chan, Y. Li, P. Lee, and Y. Xu. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *Proc. of USENIX ATC*, 2018.
- [11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.
- [12] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proc. of ACM ASPLOS*, 2021.
- [13] A. Conway, A. Gupta, V. Chidambaram, M. Farach-Colton, R. Spillane, A. Tai, and R. Johnson. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *Proc. of USENIX ATC*, 2020.
- [14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of ACM SoCC*, 2010.
- [15] Y. Dai, Y. Xu, A. Ganesan, R. Alagappan, B. Kroth, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. From Wisckey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *Proc. of USENIX OSDI*, 2020.
- [16] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal Bavigable Key-Value Store. In *Proc. of ACM SIGMOD*, 2017.
- [17] N. Dayan and S. Idreos. Dostoevsky: Better Space-Time Trade-Offs for LSM-tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proc. of ACM SIGMOD*, 2018.
- [18] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage. In *Proc. of ACM SIGMOD*, 2011.
- [19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Voshall, and W. Vogels. Dynamo: Amazon’s Highly Available Key-Value Store. In *Proc. of ACM SOSP*, 2007.
- [20] Dgraph. BadgerDB. <https://github.com/dgraph-io/badger>.
- [21] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti. Reducing DRAM footprint with NVM in Facebook. In *Proc. of ACM EuroSys*, 2018.
- [22] N. Elyasi, C. Choi, and A. Sivasubramanian. Large-Scale Graph Processing on Emerging Storage Devices. In *Proc. of USENIX FAST*, 2019.
- [23] Facebook. Memcached. <http://memcached.org>.
- [24] Facebook. RocksDB. <http://rocksdb.org/>.
- [25] Facebook. RocksDB Trace, Replay, Analyzer, and Workload Generation. <https://github.com/facebook/rocksdb/wiki/RocksDB-Trace,-Replay,-Analyzer,-and-Workload-Generation>.
- [26] Facebook. RocksDB Tuning Guide. <https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide>.
- [27] Google. LevelDB. <https://github.com/google/leveldb>.
- [28] T. Harter, D. Borthakur, S. Dong, A. S. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis of HDFS under HBase: A Facebook Messages Case Study. In *Proc. of USENIX FAST*, 2014.

- [29] J. R. Hosking and J. R. Wallis. Parameter and Quantile Estimation for the Generalized Pareto Distribution. *Technometrics*, 29(3):339–349, 1987.
- [30] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, W. Wei, C. Liu, J. Zhang, J. Li, X. Wu, L. Song, R. Sun, S. Yu, L. Zhao, N. Cameron, L. Pei, and X. Tang. TiDB: A Raft-Based HTAP Database. *Proc. of VLDB Endow.*, 13(12):3072–3084, 2020.
- [31] HyperDex. HyperLevelDB Performance Benchmarks. <http://hyperdex.org/performance/leveldb/>.
- [32] J. Im, J. Bae, C. Chung, and S. Lee. PinK: High-Speed In-Storage Key-Value Store with Bounded Tails. In *Proc. of USENIX ATC*, 2020.
- [33] X. Jiang, Y. Hu, Y. Xiang, G. Jiang, X. Jin, C. Xia, W. Jiang, J. Yu, H. Wang, Y. Jiang, J. Ma, L. Su, and K. Zeng. Alibaba Hologres: A Cloud-Native Service for Hybrid Serving/Analytical Processing. *Proc. of VLDB Endow.*, 13(12):3272–3284, 2020.
- [34] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-r. Choi. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *Proc. of USENIX FAST*, 2019.
- [35] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Redesigning LSMs for Non-volatile Memory with NoveLSM. In *Proc. of USENIX ATC*, 2018.
- [36] K. Kourtis, N. Ioannou, and I. Koltsidas. Reaping the Performance of Fast NVM Storage with uDepot. In *Proc. of USENIX FAST*, 2019.
- [37] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proc. of USENIX OSDI*, 2012.
- [38] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong. Atlas: Baidu’s Key-Value Storage System for Cloud Data. In *Proc. of IEEE MSST*, 2015.
- [39] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel. KVell: the Design and Implementation of a Fast Persistent Key-Value Store. In *Proc. of ACM SOSP*, 2019.
- [40] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu. Elasticbf: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores. In *Proc. of USENIX ATC*, 2019.
- [41] G. Lu, Y. J. Nam, and D. H. Du. BloomStore: Bloom-Filter Based Memory-Efficient Key-Value Store for Indexing of Data Deduplication on Flash. In *Proc. of IEEE MSST*, 2012.
- [42] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-Conscious Storage. In *Proc. of USENIX FAST*, 2016.
- [43] S. Luo, S. Chatterjee, R. Ketsetsidis, N. Dayan, W. Qin, and S. Idreos. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *Proc. of ACM SIGMOD*, 2020.
- [44] S. Ma, K. Chen, S. Chen, M. Liu, J. Zhu, H. Kang, and Y. Wu. ROART: Range-query Optimized Persistent ART. In *Proc. of USENIX FAST*, 2021.
- [45] F. Mei, Q. Cao, H. Jiang, and J. Li. SifrDB: A Unified Solution for Write-Optimized Key-Value Stores in Large Datacenter. In *Proc. of ACM SoCC*, 2018.
- [46] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4), 1996.
- [47] F. Pan, Y. Yue, and J. Xiong. dCompaction: Delayed Compaction for the LSM-tree. *International Journal of Parallel Programming*, 45(6):1310–1325, 2017.
- [48] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In *Proc. of USENIX ATC*, 2016.
- [49] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas. An Efficient Memory-Mapped Key-Value Store for Flash Storage. In *Proc. of ACM SoCC*, 2018.
- [50] M. Pilman, K. Bocksrocker, L. Braun, R. Marroquin, and D. Kossmann. Fast Scans on Key-Value Stores. *Proc. of VLDB Endow.*, 10(11):1526–1537, 2017.
- [51] PingCAP. Titan. <https://github.com/tikv/titan>.
- [52] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. Pebblesdb: Building Key-value Stores using Fragmented Log-Structured Merge Trees. In *Proc. of ACM SOSP*, 2017.
- [53] RedisLib. Redis. <https://redis.io>.
- [54] J. REN. YCSB-C. <https://github.com/basicthinker/YCSB-C>.
- [55] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson. SlimDB: A Space-Efficient Key-Value Storage Engine for Semi-Sorted Data. *Proc. of VLDB Endow.*, 10(13):2037–2048, 2017.
- [56] R. Sears and R. Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proc. of ACM SIGMOD*, 2012.
- [57] P. J. Shetty, R. P. Spillane, R. R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building Workload-Independent Storage with VT-trees. In *Proc. of USENIX FAST*, 2013.
- [58] X. Wu, Y. Xu, Z. Shao, and S. Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data. In *Proc. of USENIX ATC*, 2015.

- [59] J. Yang, Y. Yue, and K. Rashmi. A Large Scale Analysis of Hundreds of In-Memory Cache Clusters at Twitter. In *Proc. of USENIX OSDI*, 2020.
- [60] T. Yao, W. Jiguang, H. Ping, H. Xubin, G. Qingxin, W. Fei, , and X. Changsheng. A Light-Weight Compaction Tree to Reduce I/O Amplification Toward Efficient Key-Value Stores. In *Proc. of IEEE MSST*, 2017.
- [61] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *Proc. of USENIX ATC*, 2020.
- [62] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proc. of ACM SIGMOD*, 2018.
- [63] Q. Zhang, Y. Li, P. P. Lee, Y. Xu, Q. Cui, and L. Tang. UniKV: Toward High-Performance and Scalable KV Storage in Mixed Workloads via Unified Indexing. In *Proc. of IEEE ICDE*, 2020.
- [64] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *Proc. of USENIX FAST*, 2015.
- [65] W. Zhong, C. Chen, X. Wu, and S. Jiang. REMIX: Efficient Range Query for LSM-trees. In *Proc. of USENIX FAST*, 2021.

ZNS: Avoiding the Block Interface Tax for Flash-based SSDs

Matias Björling^{*}, Abutalib Aghayev[◇], Hans Holmberg^{*}, Aravind Ramesh^{*}, Damien Le Moal^{*},
Gregory R. Ganger[†], George Amvrosiadis[†]

^{*}Western Digital [◇]The Pennsylvania State University [†]Carnegie Mellon University

Abstract

The Zoned Namespace (ZNS) interface represents a new division of functionality between host software and flash-based SSDs. Current flash-based SSDs maintain the decades-old block interface, which comes at substantial expense in terms of capacity over-provisioning, DRAM for page mapping tables, garbage collection overheads, and host software complexity attempting to mitigate garbage collection. ZNS offers shelter from this ever-rising *block interface tax*.

This paper describes the ZNS interface and explains how it affects both SSD hardware/firmware and host software. By exposing flash erase block boundaries and write-ordering rules, the ZNS interface requires the host software to address these issues while continuing to manage media reliability within the SSD. We describe how storage software can be specialized to the semantics of the ZNS interface, often resulting in significant efficiency benefits. We show the work required to enable support for ZNS SSDs, and show how modified versions of `f2fs` and `RocksDB` take advantage of a ZNS SSD to achieve higher throughput and lower tail latency as compared to running on a block-interface SSD with identical physical hardware. For example, we find that the 99.9th-percentile random-read latency for our zone-specialized `RocksDB` is at least 2–4× lower on a ZNS SSD compared to a block-interface SSD, and the write throughput is 2× higher.

1 Introduction

The *block interface* presents storage devices as one-dimensional arrays of fixed-size logical data blocks that may be read, written, and overwritten in any order. Introduced initially to hide hard drive media characteristics and to simplify host software, the block interface worked well for many generations of storage devices and allowed great innovation on both sides of the storage device interface. There is a significant mismatch, however, between the block interface and current storage device characteristics.

For flash-based SSDs, the performance and operational costs of supporting the block interface are growing prohibitively [22]. These costs are due to a mismatch between the operations allowed and the nature of the underlying flash media. Although individual logical blocks can be written to flash, the medium must be erased at the granularity of larger

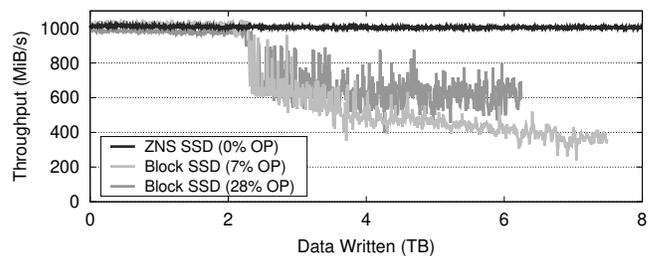


Figure 1: Throughput of a multi-threaded write workload that overwrites usable SSD capacity four times. The SSDs all have 2 TB raw media and share the same hardware platform.

units called *erase blocks*. The SSD’s Flash Translation Layer (FTL) hides this mismatch by using substantial DRAM for dynamic logical-to-physical page mapping structures, and by reserving substantial fractions (*over-provisioning*) of the drive’s media capacity to lower the garbage-collection overhead of erase-block data. Despite these efforts, garbage collection often results in throughput limitations [17], write-amplification [3], performance unpredictability [33, 64], and high tail latencies [16].

The NVMe Zoned Namespace Command Set specification [8], or ZNS for short, has recently been introduced as a new interface standard for flash-based SSDs. Instead of a single array of logical blocks, a ZNS SSD groups logical blocks into *zones*. A zone’s logical blocks can be read in random order but must be written sequentially, and a zone must be erased between rewrites. A ZNS SSD aligns zone and physical media boundaries, shifting the responsibility of data management to the host. This obviates the need for in-device garbage collection and the need for resource and capacity over-provisioning (OP). Further, ZNS hides device-specific reliability characteristics and media-management complexities from the host software.

Figure 1 illustrates some of the costs of the block interface and the corresponding potential benefits of ZNS. The three lines represent throughput for identical SSD hardware exposed as a ZNS SSD (0% OP) and as a block-interface SSD (measured with 7% OP and 28% OP), while concurrently overwriting the drive’s usable capacity 4 times. All three SSDs provide high throughput for the first 2TB (which is their raw capacity), but then the block-interface SSD expe-

periences a sharp drop in throughput as in-device garbage collection kicks in. As expected, due to the reduction in garbage collection overhead, 28% OP provides higher steady-state write throughput than 7% OP. The ZNS SSD sustains high throughput by avoiding in-device garbage collection and offers more usable capacity (see line length) by eliminating the need for over-provisioning.

This paper describes the ZNS interface and how it avoids the block interface tax (§2). We describe the responsibilities that ZNS devices jettison, enabling them to reduce performance unpredictability and significantly eliminate costs by reducing the need for in-device resources (§3.1). We also describe an expected consequence of ZNS: the host's responsibility for managing data in the granularity of erase blocks. Shifting FTL responsibilities to the host is less effective than integrating with the data mapping and placement logic of storage software, an approach we advocate (§3.2).

Our description of the consequences and guidelines that accompany ZNS adoption is grounded on a significant and concerted effort to introduce ZNS SSD support in the Linux kernel, the fio benchmark tool, the f2fs file system, and the RocksDB key-value store (§4). We describe the software modifications required in each use case, all of which have been released as open-source to foster the growth of a healthy community around the ZNS interface.

Our evaluation shows the following ZNS interface advantages. First, we demonstrate that a ZNS SSD achieves up to $2.7\times$ higher throughput than block-interface SSDs in a concurrent write workload, and up to 64% lower average random read latency even in the presence of writes (§5.1). Second, we show that RocksDB on f2fs running on ZNS SSD achieves at least $2\text{--}4\times$ lower 99.9th-percentile random read latency than RocksDB on file systems running on block-interface SSDs. RocksDB running directly on a ZNS SSD using ZenFS, a zone-aware backend we developed, achieves up to $2\times$ higher throughput than RocksDB on file systems running on block-interface SSDs (§5.2).

The paper consists of five key contributions. We present the first evaluation of a production ZNS SSD in a research paper, directly comparing it to a block-interface SSD using the same hardware platform, and optional multi-stream support. Second, we review the emerging ZNS standard and its relation to prior SSD interfaces. Third, we describe the lessons learned adapting host software layers to utilize ZNS SSDs. Fourth, we describe a set of changes spanning the whole storage stack to enable ZNS support, including changes to the Linux kernel, the f2fs file system, the Linux NVMe driver and Zoned Block Device subsystem, the fio benchmark tool, and the development of associated tooling. Fifth, we introduce ZenFS, a storage backend for RocksDB, to showcase the full performance of ZNS devices. All code changes are open-sourced and merged into the respective official codebases.

2 The Zoned Storage Model

For decades, storage devices have exposed their host capacity as a one-dimensional array of fixed-size data blocks. Through this *block interface*, data organized in a block could be read, written, or overwritten in any order. This interface was designed to closely track the characteristics of the most popular devices at the time: hard disk drives (HDDs). Over time, the semantics provided by this interface became an unwritten contract that applications came to depend on. Thus, when SSDs were introduced, they shipped with complex firmware (FTL) that made it possible to offer the same block interface semantics to applications even though they were not a natural fit for the underlying flash media.

The Zoned Storage model, originally introduced for Shingled Magnetic Recording (SMR) HDDs [19, 20, 35], was born out of the need to create storage devices free of the costs associated with block interface compatibility. We detail those costs with respect to SSDs (§2.1), and then continue to describe existing improvements to the block interface (§2.2) and the basic characteristics of zoned storage (§2.3).

2.1 The Block Interface Tax

Modern storage devices, such as SSDs and SMR HDDs, rely on recording technologies that are a mismatch for the block interface. This mismatch results in performance and operational costs. On a flash-based SSD, an empty flash page can be programmed on a write, but overwriting it requires an erase operation that can occur only at the granularity of an *erase block* (a set of one or more flash blocks, each comprising multiple pages). For an SSD to expose the block interface, an FTL must manage functionality such as in-place updates using a write-anywhere approach, mapping host logical block addresses (LBAs) to physical device pages, garbage collecting stale data, and ensuring even wear of erase blocks.

The FTL impacts performance and operational costs considerably. To avoid the media limitations for in-place updates, each LBA write is directed to the next available location. Thus, the host relinquishes control of physical data placement to the FTL implementation. Moreover, older, stale versions of the data must be garbage collected, leading to **performance unpredictability** [1, 10] for ongoing operations.

The need for garbage collection necessitates the allocation of physical resources on the device. This requires media to be over-provisioned by up to 28% of the total capacity, in order to stage data that are being moved between physical addresses. Additional DRAM is also required to maintain the volatile mappings between logical and physical addresses. Capacity over-provisioning and DRAM are the most expensive components in SSDs [18, 57], leading to **higher cost per gigabyte of usable capacity**.

2.2 Existing Tax-Reduction Strategies

Two major approaches for reducing the block interface tax have gained traction: SSDs with stream support (Stream

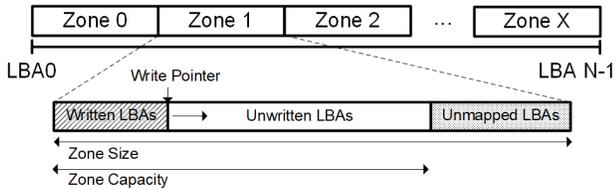


Figure 2: Zones within the LBA address space of a storage device. A write pointer per zone increases on successful writes, and is reset by issuing an explicit reset command.

SSDs) and Open-Channel SSDs (OCSSDs).

Stream SSDs allow the host to mark its write commands with a stream hint. The stream hint is interpreted by the Stream SSD, allowing it to differentiate incoming data onto distinct erase blocks [32] which *improves the overall SSD performance and media lifetime*. Stream SSDs require that the host carefully marks data with similar lifetimes in order to reduce garbage collection. If the host mixes data of different lifetimes into the same stream, Stream SSDs behave similarly to block-interface SSDs. A Stream SSD must carry the resources to manage such an event, so *Stream SSDs do not shed the costs of block-interface SSDs for extra media over-provisioning and DRAM*. We compare the performance of a Streams SSD with a ZNS SSD in §5.3.

Open-Channel SSDs allow host and SSD to collaborate through a set of contiguous LBA chunks [9, 38, 46, 60]. OCSSDs can expose these chunks such that they align with the physical erase block boundaries of the media. This eliminates in-device garbage collection overhead and reduces the cost of media over-provisioning and DRAM. With OCSSDs, the host is responsible for data placement. This includes underlying media reliability management such as wear-leveling, and specific media failure characteristics (depending on the OCSSD type). This has the potential to improve SSD performance and media lifetime over Stream SSDs, but *the host must manage differences across SSD implementations to guarantee durability*, making the interface hard to adopt and requiring continual software upkeep.

The ZNS interface, which we describe next, builds on the lessons of OCSSDs and SMR HDDs. It utilizes, and is compatible with, the zoned storage model defined in the ZAC/ZBC [28, 29] specifications. It adds features to take advantage of the characteristics of flash-based SSDs. ZNS aims to eliminate the mismatch between SSD media and the device interface. It also provides a media-agnostic next-generation storage interface by avoiding to directly manage media-specific characteristics like OCSSD [59].

2.3 Tax-free Storage with Zones

The fundamental building block of the zoned storage model is a *zone*. Each zone represents a region of the logical address space of the SSD that can be read arbitrarily but must be writ-

ten sequentially, and to enable new writes, must be explicitly reset. The write constraints are enforced by a per-zone state machine and a write pointer.

A per-zone *state machine* determines whether a given zone is writeable using the following states: EMPTY, OPEN, CLOSED, or FULL. Zones begin in the EMPTY state, transition to the OPEN state upon writes, and finally transition to FULL when fully written. The device may further impose an open zone limit on the number of zones that can simultaneously be in the OPEN state, e.g., due to device resource or media limitations. If the limit is reached and the host attempts to write to a new zone, another zone must be transitioned from the OPEN to the CLOSED state, freeing on-device resources such as write buffers. The CLOSED zone is still writeable, but must be transitioned to the OPEN state again before serving additional writes.

A zone’s *write pointer* designates the next writeable LBA within a writeable zone and is only valid in the EMPTY and OPEN states. Its value is updated upon each successful write to a zone. Any write commands issued by the host that (1) do not begin at the write pointer, or (2) write to a zone in the FULL state will fail to execute. When a zone is reset, through the reset zone command, the zone is transitioned to the EMPTY state, its write pointer is updated to the zone’s first LBA, and the previously written user data is no longer accessible. The zone’s state and write pointer eliminate the need for host software to keep track of the last LBA written to a zone simplifying recovery, e.g., after an improper shutdown.

Whilst the write constraints are fundamentally the same across zoned storage specifications, the ZNS interface introduces two concepts to cope with the characteristics of flash-based SSDs.

The *writeable zone capacity* attribute allows a zone to divide its LBAs into writeable and non-writeable, and allows a zone to have a writeable capacity smaller than the zone size. This enables the zone size of ZNS SSDs to align with the power-of-two zone size industry norm introduced with SMR HDDs. Figure 2 shows how zones can be laid out over the logical address space of a ZNS SSD.

The *active zone limit* adds a hard limit on zones that can be in either the OPEN or CLOSED state. Whereas SMR HDDs allow all zones to stay in a writeable state (i.e., CLOSED), the characteristics of flash-based media, such as program disturbs [15], require this quantity to be bounded for ZNS SSDs.

Although the ZNS interface increases the responsibilities of host software, our study shows various use cases can benefit from a set of techniques (§3.2) that ease its adoption.

3 Evolving towards ZNS

This section outlines aspects of ZNS SSDs that affect application performance, both in terms of hardware impact (§3.1) and adapting host applications to the ZNS interface (§3.2).

3.1 Hardware Impact

ZNS SSDs relinquish responsibilities traditionally carried out by the FTL, associated with supporting random writes. The ZNS interface enables the SSD to translate sequential zone writes into distinct erase blocks, thus eliminating the interface-media mismatch. Since random writes are disallowed by the interface and zones must be explicitly reset by the host, the data placement managed by the device occurs at the coarse-grained level of zones. This means that the SSD garbage collection routine responsible for moving valid data between erase blocks (to free up writeable capacity) becomes the responsibility of the host. This implies that write amplification on the device is eliminated, which eliminates the need for capacity over-provisioning, while also improving the overall performance and lifetime of the media [17, 23]. We quantify these benefits in §5.

While ZNS offers significant benefits for the end-user, it introduces the following tradeoffs in the design of the SSD's FTL.

Zone Sizing. There is a direct correlation between a zone's write capacity and the size of the erase block implemented by the SSD. In a block-interface SSD, the erase block size is selected such that data is striped across multiple flash dies, both to gain higher read/write performance, but also to protect against die-level and other media failures through per-stripe parity. It is not uncommon for SSDs to have a stripe that consists of flash blocks from 16-128 dies, which translates into a zone with writeable capacity from hundreds of megabytes to low single-digit gigabytes. Large zones reduce the degrees of freedom for data placement by the host, so we argue for the smallest zone size possible, where die-level protection is still provided, and adequate per-zone read/write performance is achieved at low I/O queue depths. If the end-user is willing to compromise on data reliability, the stripe-wide parity can be removed, and thus smaller writeable sizes can be achieved, but at the expense of host complexity, such as host-side parity calculation and deeper I/O queue depths to get the same performance as ZNS SSDs with larger zones or block-interface SSDs.

Mapping Table. In block-interface SSDs, the FTL maintains a fully-associative mapping table [21] between LBAs and their physical locations. This fine-grained mapping improves garbage collection performance, but the table size often requires 1GB of mappings per 1TB of media capacity. For consumer and enterprise SSDs, mappings are typically stored within device DRAM, whereas embedded SSDs may deploy a caching scheme at the expense of lower performance. Because ZNS SSD zone writes are required to be sequential, we can transition from complex, fully-associative mapping tables to coarse-grained mappings maintained either entirely at the erase block level [34] or in some hybrid fashion [31, 48]. As these mappings account for the largest usage of DRAM on an SSD, this can significantly reduce or even completely

eliminate the need for DRAM.

Device Resources. A set of resources is associated with each partially-written erase block (i.e., active zone). This set includes hardware resources, such as XOR engines, memory resources, such as SRAM or DRAM, and power capacitors to persist parity data following a power failure. The data and parity can range from hundreds of kilobytes to megabytes, e.g., due to two-step programming [13]. Due to these requirements and associated costs, ZNS SSDs are expected to have 8-32 active zones. Although the number of active zones can be further increased by adding extra power capacitors, utilizing DRAM for data movement, reduce parity requirements, or deploying a form of write-back cache (e.g., SLC).

3.2 Host Software Adoption

We now discuss three approaches for adapting host software to the ZNS interface. Applications that perform mostly sequential writes are prime candidates for adopting ZNS, such as Log Structure Merge (LSM) tree-based databases. Applications that primarily perform in-place updates are more challenging to support without fundamental modifications to core data structures [47].

Host-side FTL (HFTL). An HFTL acts as a mediator between (1) a ZNS SSD's write semantics and (2) applications performing random writes and in-place updates. The HFTL layer is similar to the responsibilities of the SSD FTL, but the HFTL layer manages only the translation mapping and associated garbage collection. Although it has less responsibility than an SSD FTL, an HFTL must manage its utilization of CPU and DRAM resources because it shares them with host applications. An HFTL makes it easier to integrate host-side information and enhances control of data placement and garbage collection, while also exposing the conventional block interface to applications. Existing work, such as dm-zoned [44], dm-zap [24], pblk [9], and SPDK's FTL [40], shows the feasibility and applicability of an HFTL, but only dm-zap currently supports ZNS SSDs.

File Systems. Higher-level storage interfaces (such as the POSIX file system interface) enable multiple applications to access storage by means of common file semantics. By integrating zones with higher layers of the storage stack, i.e., ensuring a primarily sequential workload, we eliminate the overhead that is otherwise associated with both HFTL and FTL data placement [30], as well as the indirection overhead [64] associated with them. This also allows additional data characteristics known to higher storage stack layers to be used to improve on-device data placement (at least to the degree that the application's actual workload allows this information to permeate to such layers).

The bulk of the file systems developed today primarily perform in-place writes and are generally difficult to adapt to the Zoned Storage model. Some file systems, however, such as f2fs [36], btrfs [53], and zfs [41] exhibit overly-sequential

write patterns and are already adapting so that they can support zones [4, 43]. Although not all of their writes are sequential (such as superblock and some metadata writes), file systems like these can be extended with strict log-structured writes [43], a floating superblock [4], and similar functionality to bridge the gap. These file systems effectively mimic the HFTL logic (with the LBA mapping table managed on-disk through metadata) while also implementing garbage collection to defragment data and free up space for new writes. Although zone support exists for f2fs and btrfs, they support only the zone model that is defined in ZAC/ZBC. As part of this work, we implement the necessary changes to f2fs to showcase the relative ease of supporting ZNS’s zone model, and evaluate its performance (§4.1).

End-to-End Data Placement. In an ideal world, zone-write semantics would be aligned with an application’s existing data structures. This would allow the highest degree of freedom by enabling the application to manage data placement, while at the same time eliminating indirection overheads from file-system and translation layers. While end-to-end data placement enables a collaboration between the application and ZNS SSD, and has the potential to achieve the best write amplification, throughput, and latency improvements, it is as daunting as interacting with raw block devices.

File semantics are a useful abstraction, and by forgoing them one must not only integrate zone support, but also provide tools for the user to perform inspection, error checking, and backup/restore operations. Like file systems, applications that exhibit sequential write patterns are prime candidates for end-to-end integration. This includes LSM-tree-based stores such as RocksDB, caching-based stores such as CacheLib [7], and object stores such as Ceph SeaStore [55]. To showcase the benefits of end-to-end integration, we introduce ZenFS, a new RocksDB zoned storage backend and compare it to both (1) the XFS file system and (2) the f2fs file system, with and without integrated ZNS support.

4 Implementation

We have added support to four major software projects to evaluate the benefits of ZNS. First, we made modifications to the Linux kernel to support ZNS SSDs. Second, we modified the f2fs file system to evaluate the benefits of zone integration at a higher-level storage stack layer. Third, we modified the fio [6] benchmark to support the newly added ZNS-specific attributes. Fourth, we developed ZenFS [25], a novel storage backend for RocksDB that allows control of data placement through zones, to evaluate the benefits of end-to-end integration for zoned storage. We describe the relatively few changes necessary to support ZNS when building upon the existing ZAC/ZBC support for the first three projects [5, 42, 52] (§4.1) and finally detail the architecture of ZenFS (§4.2).

Table 1 shows the lines modified for each software project. All the modifications have been contributed and accepted into the respective codebases [12, 25, 50, 51].

Project	Lines Added	Lines Removed
Linux Kernel	647	53
f2fs (kernel)	275	37
f2fs (mkfs tool)	189	15
fio	342	58
ZenFS (RocksDB)	3276	2
Total	4729	165

Table 1: Lines modified across projects to add ZNS support.

4.1 General Linux Support

The Linux kernel’s Zoned Block Device (ZBD) subsystem is an abstraction layer that provides a single unified zoned storage API on top of various zoned storage device types. It provides both an in-kernel API and an ioctl-based user-space API supporting device enumeration, report of zones, and zone management (e.g., zone reset). Applications such as fio utilize the user-space API to issue I/O requests that align with the write characteristics of the underlying zoned block device, regardless of its low-level interface.

To add ZNS support to the Linux kernel and the ZBD subsystem, we modified the NVMe device driver to enumerate and register ZNS SSDs with the ZBD subsystem. To support the ZNS SSD under evaluation, the ZBD subsystem API was further extended to expose the per zone capacity attribute and the active zones limit.

Zone Capacity. The kernel maintains an in-memory representation of zones (a set of zone descriptor data structures), which is managed solely by the host unless errors occur, in which case the zone descriptors should be refreshed from the specific disk. The zone descriptor data structure is extended with a new zone capacity attribute and versioning, allowing host applications to detect the availability of this new attribute.

Both fio and f2fs are updated to recognize the new data structure. Whereas fio simply had to avoid to issue write I/Os beyond the zone capacity, f2fs required additional changes.

f2fs manages capacity at the granularity of segments, typically 2MiB chunks. For zoned block devices f2fs manages multiple segments as a section, the size of which is aligned to the zone size. f2fs writes sequentially across a section’s segments, and partially writeable zones are not supported. To add support for the zone capacity attribute in f2fs, the kernel implementation, and associated f2fs-tools, two extra segment types are added to the three conventional segment types (i.e., free, open, and full): an *unusable* segment type that maps the unwriteable part of a zone, and a *partial* segment type that covers the case where a segment’s LBAs cross both the writeable and the unwriteable LBAs of a zone. The partial segment type explicitly allows optimizing for the case when segment chunk size and the zone capacity of a specific zone are unaligned, utilizing all of the writeable capacity of a zone.

To allow backward compatibility for SMR HDDs, which do not expose zone capacity, the zone capacity is initialized

to the zone size attribute.

Limiting Active Zones. Due to the nature of flash-based SSDs, there is a strict limit on the number of zones that can be active simultaneously, i.e., either in OPEN or CLOSED state. The limit is detected upon zoned block device enumeration, and exposed through the kernel and user-space APIs. SMR HDDs do not have such a limit and the attribute is initialized to zero (read: infinity).

No modifications were made to fio to support the limit, so it is the responsibility of the fio user to respect the active-limit constraint, which otherwise results in an I/O error when no more zones can be opened.

For f2fs, the limit is linked to the number of segments that can be open simultaneously. f2fs limits this to a maximum of 6, but can be reduced to align with the available active zone limit. This limit is set at the time the file system is created, and f2fs-tools is modified to check for the zone active limit, and if the zoned block device does not support enough active zones, the number of open segments is configured to align with what is available by the device.

f2fs requires its metadata to be stored on a conventional block device, necessitating a separate device. We do not directly address this in our modifications, as the evaluated ZNS SSD exposes a fraction of its capacity as a conventional block device. If this is not supported by the ZNS SSD, write-in-place functionality can be added similar to btrfs [4], or a small translation layer [24] can expose a limited set of zones on the ZNS SSD through the conventional block interface. Note that the Slack Space Recycling (SSR) feature (i.e., random writes) is disabled for all zoned storage devices, which decreases the overall performance. However, due to overall higher performance achieved by ZNS SSDs, we demonstrate superior performance to block-interface SSDs which have SSR enabled (§5.2).

4.2 RocksDB Zone Support

In this section we show how to adapt the popular key-value database RocksDB to perform end-to-end data placement onto zoned storage devices using the ZenFS storage backend. ZenFS takes advantage of the log-structured merge (LSM) tree [45] data structure of RocksDB that it uses to store and maintain its data, and its associated immutable sequential-only compaction process.

The LSM tree implementation consists of multiple levels, as shown in Figure 3. The first level (L_0) is managed in-memory and flushed to the level below periodically or when full. Intermediate updates between flushes are made durable using a Write-Ahead Log (WAL). The rest of the levels (L_1, \dots, L_n) are maintained on-disk. New or updated key-value data pairs are initially appended to L_0 , and upon flush the key-value data pairs are sorted by key, and then written to disk as a Sorted String Table (SST) file.

The size of a level is typically fitted to a multiple of the level above, and each level contains multiple SSTs with each

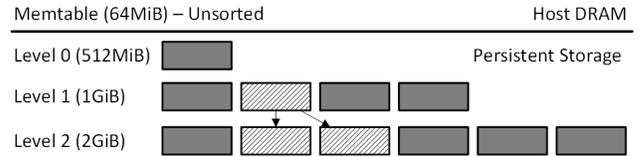


Figure 3: Data organization in RocksDB. Dark grey squares represent SSTs. Light grey with lines squares represent SSTs selected for compaction.

SST containing an ordered set of non-overlapping key-value data pairs. Through an explicit compaction process, an SST's key-value data pairs are merged from one level (L_i) to the next (L_{i+1}). The compaction process reads key-value data pairs from one or more SSTs and merges them with the key-value pairs from one or more SSTs in the next level. The merged result is stored in a new SST file and replaces the merged SSTs in the LSM tree. As a result of this process, SST files are immutable, written sequentially, and created/removed as a single unit. Furthermore, hot/cold data separation is achieved as key-value data pairs are merged into levels below.

RocksDB has support for separate storage backends through its file system wrapper API that is a unified abstraction for RocksDB to access its on-disk data. At its core, the wrapper API identifies data units, such as SST files or Write-Ahead Log (WAL), through a unique identifier (e.g., a filename) that maps to a byte-addressable linear address space (e.g., a file). Each identifier supports a set of operations (e.g., add, remove, current size, utilization) in addition to random access and sequential-only byte-addressable read and write semantics. These are closely related to file system semantics, where identifier and data is accessible through files, which is RocksDB's main storage backend. By using a file system that manages files and directories, RocksDB avoids managing file extents, buffering, and free space management, but also loses the ability to place data directly into zones, which prevents end-to-end data placement onto zones, and therefore reduces the overall performance.

4.2.1 ZenFS Architecture

The ZenFS storage backend implements a minimal on-disk file system and integrates it using RocksDB's file-wrapper API. It carefully places data into zones while respecting their access constraints, and collaborates with the device-side zone metadata on writes (e.g., write pointer), reducing complexity associated with durability. The main components of ZenFS are depicted in Figure 4 and described below.

Journaling and Data. ZenFS defines two types of zones: journal and data zones. The journal zones are used to recover the state of the file system, and maintains the superblock data structure, and mapping of WAL and data files to zones, whereas the data zones store the file content.

Extents. RocksDB's data files are mapped and written to a

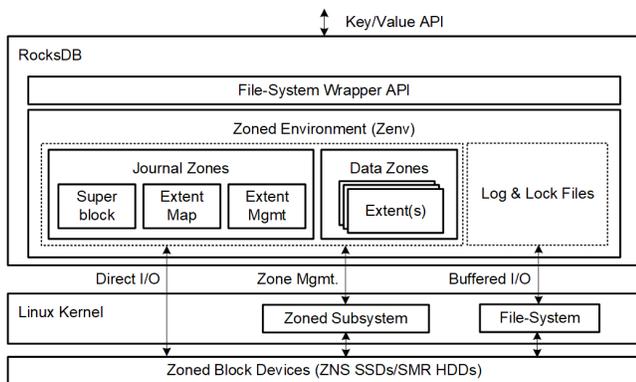


Figure 4: ZenFS Architecture

set of extents. An *extent* is a variable-sized, block-aligned, contiguous region that is written sequentially to a data zone, containing data associated to a specific identifier. Each zone can store multiple extents, but extents do not span zones. Extent allocation and deallocation events are recorded in an in-memory data structure and written to the journal when a file is closed or the data is requested to be persisted by RocksDB through an `fsync` call. The in-memory data structure keeps track of the mapping of extents to zones, and once all files with allocated extents in a zone has been deleted the zone can be reset and reused.

Superblock. The superblock data structure is the initial entry point when initializing and recovering ZenFS state from disk. The superblock contains a unique identifier for the current instance, magic value, and user options. A unique identifier (UUID) in the superblock allows the user to identify the file-system even if the order of block device enumeration on the system changes.

Journal. The responsibility of the journal is to maintain (1) the superblock and (2) the WAL and data file to zone translation mapping through extents.

The journal state is stored on dedicated journal zones, and is located on the first two non-offline zones of a device. At any point in time, one of the zones are selected as the active journal zone, and is the one that persists updates to the journal state. A journal zone has a header that is stored at the beginning of the specific zone. The header stores a sequence number (incremented each time a new journal zone is initialized), the superblock data structure, and a snapshot of the current journal state. After the header are stored, the remaining writable capacity of the zone is used to record updates to the journal.

To recover the journal state from disk, three steps are required: ① the first LBA for each of the two journal zones must be read to determine the sequence number of each, where the journal zone with the highest value is the current active zone; ② the full header of the active zone is read and initializes the initial superblock and journal state; and finally ③

SST file size	128 MiB	256 MiB	512 MiB
Write Amp.	11.9×	12.0×	12.0×
Runtime (s)	15,430	15,461	14,918
99.99 RW lat (ms)	102	102	105
99.99 RWL lat (ms)	77	73	73

Table 2: RocksDB’s write amplification and runtime during the *fillrandom* benchmark, and 99.99-th percentile tail latencies during the *readwhilewriting* benchmark with no rate limits (RW) and with writes rate-limited to 20 MiB/s (RWL), with different SST file sizes.

journal updates are applied to the header’s journal snapshot.

The amount of updates to apply is determined by the zone’s state, and its write pointer. If the zone is in the OPEN (or CLOSED) state, only records up to the current write pointer value are replayed to the journal, whereas if the zone is in the FULL state, all records stored after the header are replayed. Note that if the zone is full, after recovery a new active journal zone is selected and initialized to enable persisting journal updates.

The initial journal state is created and persisted by an external utility that is similar to existing file system tools that generate the initial on-disk state of a file system. It writes the initial sequence number, superblock data structure and an empty snapshot of the journal to the first journal zone. When ZenFS is initialized by RocksDB, the above recovery process is executed, after which it is ready for data accesses from RocksDB.

Writeable Capacity in Data Zones. Ideal allocation, resulting in maximum capacity usage over time, can only be achieved if file sizes are a multiple of the writeable capacity of a zone allowing file data to be completely separated in zones while filling all available capacity. File sizes can be configured in RocksDB, but the option is only a recommendation and sizes will vary depending on the results of compression and compaction processes, so exact sizes are not feasible. ZenFS addresses this by allowing a user-configurable limit for finishing data zones, specifying the percentage of the zone capacity remaining. This allows the user to specify a file size recommendation of, e.g., 95% of device’s zone capacity by setting the finish limit to 5%. This allows for the file size to vary within a limit and still achieve file separation by zones. If the file size variation is outside the specified limit, ZenFS will make sure that all available capacity is utilized by using its zone allocation algorithm (described below). Zone capacities are generally larger than the RocksDB recommended file size of 128 MiB and to make sure that increasing the file size does not increase RocksDB write amplification and read tail latencies we measured the impact on different file sizes. Table 2 shows that increasing SST file sizes does not significantly reduce performance.

Data Zone Selection. ZenFS employs a best-effort algorithm to select the best zone to store RocksDB data files. RocksDB separates the WAL and the SST levels by setting a write-lifetime hint for a file prior to writing to it. Upon the first write to a file, a data zone is allocated for storage. ZenFS tries first to find a zone based on the lifetime of the file and the max lifetime of the data stored in the zone. A match is only valid if the lifetime of the file is less than the oldest data stored in the zone to avoid prolonging the life of the data in the zone. If several matches are found, the closest match is used. If no matches are found, an empty zone is allocated. If the file fills up the remaining capacity of the allocated zone, another zone is allocated using the same algorithm. Note that the write life time hint is provided to any RocksDB storage backend, and is therefore also passed to other compatible file systems and can be used together with SSDs with stream support. We compare both approaches to pass hints in §5.3. *By using the ZenFS zone selection algorithm and the user-defined writeable capacity limits, the unused zone space or space amplification is kept at around 10%.*

Active Zone Limits. ZenFS must respect the active zone limits specified by the zoned block device. To run ZenFS, a minimum of three active zones are required, which are separately allocated to the journal, WAL, and compaction process. To improve performance, the user can control the number of concurrent compactions. Our experimentation has shown that by limiting the number of concurrent compactions, *RocksDB can work with as few as 6 active zones with restricted write performance*, while more than 12 active zones does not add any significant performance benefits.

Direct I/O and Buffered Writes. ZenFS leverages the fact that writes to SST files are sequential and immutable and performs direct I/O writes for SST files, bypassing the kernel page cache. For other files, such as the WAL, ZenFS buffers writes in memory and flushes the buffer when it is full, the file is closed, or when RocksDB requests a flush. If a flush is requested, the buffer is padded to the next block boundary and an extent with the number of valid bytes is stored in the journal. The padding results in a small amount of write amplification, but it is not unique to ZenFS, and is similarly done in conventional file system.

5 Evaluation

To evaluate the benefits and compare the performance of ZNS SSDs, we utilize a production SSD hardware platform that can expose itself as either a block-interface SSD or a ZNS SSD. As such, we enable an apples-to-apples comparison between the two interfaces. The block-interface SSD is formatted to have either 7% OP or 28% OP, and supports streams. ZNS SSD-specific details are shown in Table 3.

Our evaluation of ZNS is constructed around evaluating the following aspects:

- **Raw device I/O performance (§5.1).** We perform an

SSD Interface	Conv.	Conv.	ZNS
Media Capacity	2 TiB	2 TiB	2 TiB
Host Capacity	1.92 TB	1.60 TB	2 TB
Over-provisioning	7%	28%	0%
Placement Type	None	None	Zones
Max Active Zones	N/A	N/A	14
Zone Size	N/A	N/A	2048 MiB
Zone Capacity	N/A	N/A	1077 MiB

Table 3: Feature summary of the evaluated SSDs

apples-to-apples comparison between the block-interface SSDs and the ZNS SSD. We find that the ZNS SSD achieves higher concurrent write throughput and lower random read latency in the presence of writes.

- **End-to-end application performance (§5.2).** We compare the performance of RocksDB on a block-interface SSD running the XFS and f2fs file systems, with the performance of RocksDB when running on a ZNS SSD using our ZenFS backend. We find that RocksDB achieves up to 2× higher read and write throughput, and an order of magnitude lower random read tail latency when running over the ZNS SSD.
- **End-to-end performance comparison with SSD Streams (§5.3).** We compare the performance of the ZNS SSD with a Streams-enabled block-interface SSD, by running RocksDB on XFS and f2fs. We find that RocksDB achieves up to 44% higher throughput on ZNS and up to half the tail latency compared to the Stream SSD.

All experiments are performed on a Dell R7515 system with a 16-core AMD Epyc 7302P CPU and 128 GB of DRAM (8×16GB DDR4-3200Mhz). The baseline system configuration is an Ubuntu 20.04 distribution, updated to the 5.9 Linux kernel—the first version that includes our contributions as described in §4.1. RocksDB experiments were carried out on RocksDB 6.12 with ZenFS included as a backend storage plugin.

5.1 Raw I/O Characteristics

To ensure that the same workload is applied to both the block-interface SSD and ZNS SSDs, we divide the block-interface SSD address space into LBA ranges that have the same number of LBAs as the zone capacity exposed by the ZNS SSD. These zones, or LBA ranges, respect the same sequential write constraint as ZNS, but the zone reset is simulated by trimming the LBA range before writing. We measure performance after the SSD has reached its steady-state with a given workload.

Sustained Write Throughput. We evaluate SSD throughput to show the internal SSD garbage collection impact on throughput and its ability to consume host writes. The experiment issues 64KiB write I/Os to 4 zones. When one zone

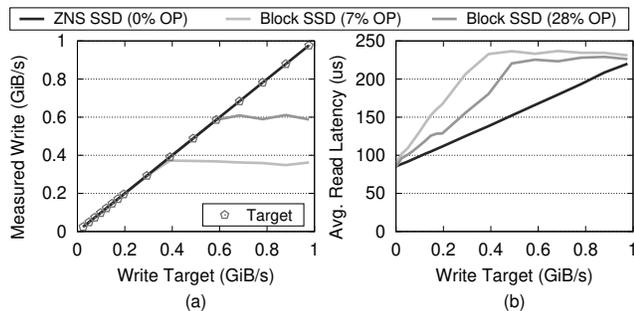


Figure 5: Measured (a) write throughput and (b) average random read latency during rate-limited writes.

is full, a new zone is chosen, reset, and written sequentially. While running the experiment, we measure the drive’s ability to reach a specific host write throughput target ranging from 0 to 1GiB/s. Figure 5 (a) shows the achieved write throughput for each write target. The block-interface SSDs sustain target writes up to 300MiB/s (0% OP) and 500MiB/s (28% OP), whereas the ZNS SSD sustains 1GiB/s. This aligns with the measured steady-state throughput shown in Figure 1, which shows that the block-interface SSDs achieve 370MiB/s (7% OP) and 590MiB/s (28% OP), respectively. The ZNS SSD with 0% OP, however, reaches 1010MiB/s. This is 1.7 – 2.7 \times higher write throughput, and 7 – 28% more storage capacity which is no longer required by the device-side garbage collection.

The ZNS SSD achieves higher throughput and lower write amplification as it can align writes onto distinct erase blocks and avoid garbage collection. The block-interface SSD mixes data onto the same erase block, which is ultimately garbage collected at separate points in time, increasing garbage collection overhead.

Random Reads and Writes. To demonstrate the reduced latency of random reads on a ZNS SSD, a second process is added to the previous experiment, which simultaneously performs random 4 KiB read I/Os across the SSD. We measure its average read latency while gradually increasing the host write target. Figure 5 (b) shows the average random read latency of the block-interface SSDs and the ZNS SSD. With no ongoing writes, both block-interface SSDs report a 4KiB read latency of 85 μ s. Compared to the SSDs with 7% and 28% OP, the random latency of the ZNS SSD is 19% and 5% lower at 50MiB/s write throughput, 45% and 16% lower at 150MiB/s write throughput, and finally 64% and 27% lower at 300MiB/s write throughput. The ZNS SSD’s throughput continues to increase linearly with the write target, unlike the block-interface SSDs.

5.2 RocksDB

Next, we demonstrate the performance improvements achieved by RocksDB when it runs on a ZNS SSD accessed either through f2fs with ZNS support added, or on top of our

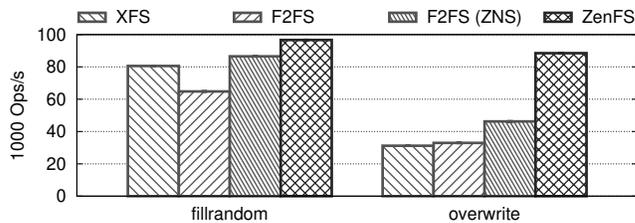


Figure 6: Throughput of RocksDB with write-heavy benchmarks— *fillrandom* followed by *overwrite* using the block-interface SSD with 28% OP and the ZNS SSD.

end-to-end optimized storage backend ZenFS. We compare these two setups against RocksDB running on XFS and f2fs using the block-interface SSD. f2fs supports both storage interfaces, allowing us to compare the impact of a file system running on top of a ZNS SSD. Furthermore, ZenFS is optimized for ZNS, showing the benefits of integrating data placement into an I/O intensive application.

Five workloads are executed for each setup. *fillrandom* is a write-intensive workload pre-conditioning the SSDs for follow-on benchmarks by issuing writes to fill drive capacity several times. Second, *overwrite* measures the overall performance when overwriting. Third, *readwhilewriting* performs random reads while writing. Fourth, *random reads* performs as described. Finally, *readwhilewriting* performs as described with the addition of write rate-limiting. The last three benchmarks are ran three times each and their results are averaged.

The benchmarks are scaled to 80% of the SSD. As a result, the actual over-provisioned space is 27% and 48% with respect to 7% OP and 28% OP. In the interest of space, we show mainly the 28% results, but show the performance difference for the write-heavy benchmarks and the rate-limited *readwhilewriting* benchmark. A 128 MiB target SST file size is used in all benchmarks except for the ZenFS workloads where the target file size is configured to align with the zone capacity.

Write Throughput. We first study the write throughput improvements, using two write-heavy benchmarks on the SSD with 28% OP and the ZNS SSD. First, we populate the database using the *fillrandom* benchmark with 3.8 billion 20B keys and 800B values (compressed to 400B). Then, we randomly overwrite all values using the *overwrite* benchmark.

Figure 6 shows the average throughput of the two benchmarks over the four setups. In the *fillrandom* benchmark, we report the average of write operations per second (ops/s) for each run and is executed from a clean state. Thus, the result is impacted by the lower garbage collection overhead until the SSD reaches a steady-state. As a result, the full write amplification impact can be first seen in the *overwrite* benchmark. Both benchmarks show the XFS and f2fs setups perform lower than f2fs (ZNS) and ZenFS. The most significant impact is seen in the *overwrite* benchmark, in which the garbage collection overhead heavily impacts the overall performance.

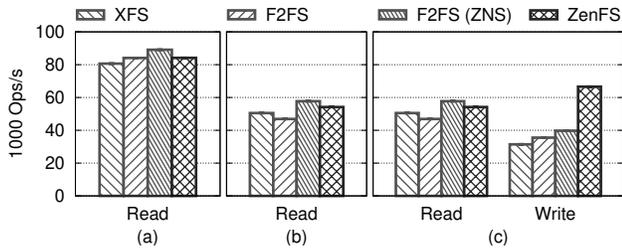


Figure 7: Throughput of RocksDB reads during (a) the *randomread* benchmark, (b) the *readwhilewriting* benchmark with writes rate-limited to 20 MiB/s and (c) the *readwhilewriting* benchmark with no write limits using the block-interface SSD with 28% OP and the ZNS SSD.

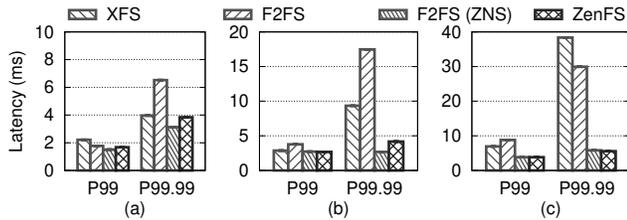


Figure 8: Latency of RocksDB reads during the (a) *randomread* benchmark, (b) the *readwhilewriting* benchmark with writes rate-limited to 20 MiB/s and (c) *readwhilewriting* benchmark with no write limits using the block-interface SSD with 28% OP and the ZNS SSD.

ZenFS is 183% faster than XFS, and f2fs (ZNS) performs 42% better than XFS and 33% better than f2fs.

While the performance of f2fs ZNS increases, the garbage collection overhead associated to large section size requires more data to be moved upon cleaning, which ultimately impacts the host-side garbage collection. To confirm, we measure the write amplification factor on the block-interface SSD, which is reported as $2.0\times$ for XFS, and $2.4\times$ for f2fs, whereas no device-side garbage collection (i.e., $1.0\times$) occurred for f2fs (ZNS) and ZenFS.

Read While Writing. As we have already shown (§5.1), the ZNS SSD achieves lower read latency during concurrent writes because there are no in-device garbage collection operations to interfere with reads during writes. We now demonstrate how this affects the latency of RocksDB read throughput using two read-intensive benchmarks: *randomread* and *readwhilewriting*. The first initiates 32 threads that perform a random reads of key-value pairs on a drive that has been pre-conditioned by the write-intensive benchmarks. The second initiates an extra thread that performs random overwrites, in addition to the 32 threads performing reads.

First, we run the *randomread* benchmark, next we run *readwhilewriting* while rate-limiting writes to 20 MiB/s (i.e., 25.6 Kops/s), and finally we run *readwhilewriting* with no rate-limiting of writes. We run each of these benchmarks for 30 minutes, two times, again in the same four setups as before,

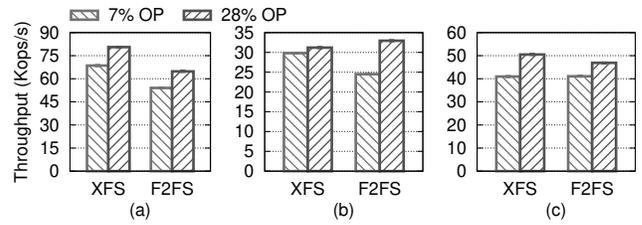


Figure 9: Throughput of RocksDB writes during (a) *fillrandom*, (b) *overwrite*, and (c) *readwhilewriting* benchmark, using the block-interface SSD with 7% OP, as well as 28% OP.

and report read/write throughput in Figure 7 and the average and tail read latencies per operation in Figure 8.

For the *randomread* benchmark, we observe that each combination achieves similar number of write ops/s and average latency, which is expected as there is no writes ongoing. However, the 99th percentile (P99) latencies show that ZenFS is 25% lower than XFS, and 6% lower than f2fs, while f2fs (ZNS) has 32% lower latency than XFS and 16% lower than f2fs.

Next, we study the throughput when performing rate-limited writes. We first notice that only ZenFS is able to sustain 20MiB/s writes while reading, with the other file systems are doing 15% less writes. f2fs (ZNS) performs the most read ops/s, followed by ZenFS, f2fs and XFS. Furthermore, we observe that ZenFS and f2fs (ZNS) have the lowest average latencies, and are able to achieve P99.9 read latencies that are $2\times$ and $4\times$ lower than for XFS and f2fs, respectively.

Finally, we remove the write rate-limit, and then measure the overall impact. Removing the write limit allows ZenFS to achieve $2\times$ the write ops/s compared to f2fs and XFS on consumer SSDs at higher level of read ops/s. f2fs (ZNS) achieves the highest amount of read ops/s and significantly higher write throughput than XFS and f2fs. The P99.99 read tail latencies stand out for ZenFS and f2fs (ZNS) which are an order of magnitude lower than f2fs and XFS.

Impact of Capacity Over-Provisioning. We also evaluate the impact of SSD capacity over-provisioning on RocksDB performance. Whereas block-interface SSDs thrive on unused capacity, as garbage collection is improved and thereby lowering its write amplification factor. The ZNS SSD do not exhibit this behavior and can use all available capacity, while still maintaining a write amplification factor of $\sim 1\times$.

Figure 9 shows the measured number of operations per second for XFS and f2fs when executing the *fillrandom*, *overwrite*, and rate-limited *readwhilewriting* benchmarks. The benchmarks are executed with 7% OP and 28% OP, respectively. For the *fillrandom* benchmark, an improvement between 14% and 16% in overall operations is seen, whereas in the *overwrite* benchmark, XFS only sees a modest 4% improvement, while f2fs is improved by 25%. Finally, in the *readwhilewriting* benchmark we see an improvement between

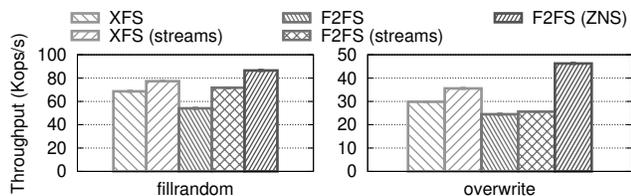


Figure 10: The throughput of RocksDB during the *fillrandom* and *overwrite* benchmarks when executing on an block-interface SSD with 7% OP and stream support.

12% and 18% for f2fs and XFS. While the overall throughput is important, the latency is as well. For XFS, the tail latency improves by 30% and by 23% for P99 and P99.9. Thus, if latency is more important than throughput, then one must budget with addition over-provisioning to lower the overall latency caused by extra garbage collection.

5.3 Streams

Finally, we evaluate the performance of a block-interface SSD with stream support against the performance of a ZNS SSD.

Figure 10 shows the throughput of the *fillrandom* and *overwrite* benchmarks executing on top of a block-interface SSD with 7% OP and with streams enabled or disabled on top of XFS and f2fs. RocksDB on XFS shows higher throughput by 11% and 16% with Streams on *fillrandom* and *overwrite* respectively. RocksDB on f2fs shows higher throughput by 24% and 4% with Streams on *fillrandom* and *overwrite* respectively. f2fs (ZNS) achieves 17% and 44% higher throughput compared to f2fs on a block-interface SSD with Streams for *fillrandom* and *overwrite* respectively.

Furthermore, the P99 latency for f2fs drops from 9,435 μ s to 6,529 μ s with Streams support. This is very close to ZenFS’s P99 latency during the same benchmark of 3,734 μ s. We therefore find that RocksDB achieves up to 44% higher throughput on ZNS and close to half the tail latency compared to a block-interface SSD with streams [56].

6 Related Work

This section covers related work on host-device collaboration, research platforms, and key-value store designs. This section covers work that was not already covered as necessary background (§2.2).

Host-Device Collaboration. Significant research has gone into optimizing the storage interface between host and SSD. Josephson et al. implement DFS [30], a host-side file system tightly integrated with the underlying hardware. SDF [46] exposes individual flash dies to the host through a custom-designed storage controller, and FlashBlox [26] uses dedicated channels and dies for each application to improve isolation. Application-Managed Flash [38] defines an interface based on fixed-size segments that can be written sequentially and reset by a trim operation. ZNS does not fix the number of erase blocks per zone, which allows the SSD implementation

to map erase blocks dynamically to zones. Zhang et al. examine sharing the responsibility of data placement by enabling the SSD to notify the host upon data relocation events [67]. ZNS provides a path to have both high performance and low cost, while leaving the reliability and coarse-grained management to the device.

Research Platforms. Various SSD hardware [14, 37, 58] and software [11, 39, 66] platforms have been developed over the years to expose the characteristics inherent to SSDs [10, 23]. This enables key research on storage interfaces [30, 54, 63] and makes it possible to improve upon the block device interface [27, 62]. Each of these works target specific optimizations and platforms, enabling the storage community to build on them. ZNS is built upon these advancements, each of which has enabled further exploration of the block interface design space.

Key-value Store Designs. Previous work introduces multiple key-value store designs for non-block interfaces. LOCS [61] takes end-to-end design to the extreme and modifies LevelDB to leverage the channel parallelism in OCSSDs. Unlike LOCS, which is tied to a specific OCSSD platform, ZenFS targets the general zone interface and thereby enables RocksDB to run on both ZNS SSDs and HM-SMR HDDs. Similarly, SMRDB [49], GearDB [65], and BlueFS [2] target strictly HM-SMR HDDs and are therefore unable to take the advantage of ZNS extensions and run on ZNS SSDs.

Conclusion

ZNS enables higher performance and lower-cost-per-byte flash-based SSDs. By shifting responsibility for managing data organization within erase blocks from FTLs to host software, ZNS eliminates in-device LBA-to-page maps, garbage collection and over-provisioning. Our experiments with ZNS-specialized f2fs and RocksDB implementations show substantial improvements in write throughput, read tail latency, and write-amplification when compared to conventional FTLs running on identical SSD hardware.

Acknowledgments

We thank the 22 anonymous reviewers, and our shepherd Ethan Miller for their help improving the presentation of this paper. We thank the members and companies of the PDL Consortium: Amazon, Facebook, Google, Hewlett Packard Enterprise, Hitachi Ltd., IBM Research, Intel Corporation, Microsoft Research, NetApp, Inc., Oracle Corporation, Pure Storage, Salesforce, Samsung Semiconductor Inc., Seagate Technology, Two Sigma, and Western Digital for their interest, insights, feedback, and support. This work has been made possible in part by gifts from VMware’s University Research Fund and from the NetApp University Research Fund.

References

- [1] Abutalib Aghayev and Peter Desnoyers. Skylight—A Window on Shingled Disk Operation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 135–149, Santa Clara, CA, USA, February 2015. USENIX Association.
- [2] Abutalib Aghayev, Sage Weil, Greg Ganger, and George Amvrosiadis. Reconciling LSM-Trees with Modern Hard Drives using BlueFS. Technical Report CMU-PDL-19-102, CMU Parallel Data Laboratory, April 2019.
- [3] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design Tradeoffs for SSD performance. In *USENIX Annual Technical Conference*, volume 57. Boston, USA, 2008.
- [4] Naohiro Aota. btrfs: Zoned block device support. <https://lwn.net/Articles/836726/>, 2020.
- [5] Bart Van Assche. fio: Add zoned block device support. <https://github.com/axboe/fio/pull/585>, 2018.
- [6] Jens Axboe. Flexible I/O Tester. <git://git.kernel.dk/fio.git>, 2016.
- [7] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosf, 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 20)*, pages 753–768, 2020.
- [8] Matias Bjørling. NVM Express Zoned Namespaces Command Set 1.0, June 2020. Available from <http://www.nvmexpress.org/specifications>.
- [9] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, Santa Clara, CA, 2017. USENIX Association.
- [10] Luc Bouganim, Bjorn Jónsson, and Philippe Bonnet. uFLIP: Understanding Flash IO Patterns. In *Proceedings of the Int’l Conf. on Innovative Data Systems Research (CIDR)*, Asilomar, California, USA, 2009.
- [11] John S Bucy, Gregory R Ganger, et al. *The DiskSim simulation environment version 3.0 reference manual*. School of Computer Science, Carnegie Mellon University, 2003.
- [12] Keith Busch, Hans Holmberg, Ajay Joshi, Aravind Ramesh, Niklas Cassel, Matias Bjørling, Damien Le Moal, and Keith Busch. Linux kernel Zoned Namespace support. <https://lwn.net/ml/linux-block/20200615233424.13458-6-keith.busch@wdc.com/>, 2020.
- [13] Yu Cai, Saugata Ghose, Yixin Luo, Ken Mai, Onur Mutlu, and Erich F Haratsch. Vulnerabilities in MLC NAND flash memory programming: experimental analysis, exploits, and mitigation techniques. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 49–60. IEEE, 2017.
- [14] Yu Cai, Erich F Haratsch, Mark McCartney, and Ken Mai. FPGA-based Solid-state Drive Prototyping Platform. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 101–104. IEEE, 2011.
- [15] Yu Cai, Onur Mutlu, Erich F Haratsch, and Ken Mai. Program Interference in MLC NAND Flash Memory: Characterization, Modeling, and Mitigation. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 123–130. IEEE, 2013.
- [16] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56:74–80, 2013.
- [17] Peter Desnoyers. Analytic Modeling of SSD Write Performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*, page 12. ACM, 2012.
- [18] DRAMeXchange. NAND Flash Spot Price, September 2014. <http://dramexchange.com>.
- [19] Tim Feldman and Garth Gibson. Shingled Magnetic Recording: Areal Density Increase Requires New Data Management. *login*, 38(3), June 2013.
- [20] Garth Gibson and Greg Ganger. Principles of Operation for Shingled Disk Devices. Technical Report CMU-PDL-11-107, CMU Parallel Data Laboratory, April 2011.
- [21] Aayush Gupta, Youngjae Kim, and Bhuvan Urganekar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. *ACM SIGPLAN Notices*, 44(3):229, 2009.
- [22] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 263–276, Santa Clara, CA, 2016. USENIX Association.

- [23] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the twelfth European conference on computer systems*, pages 127–144. ACM, 2017.
- [24] Hans Holmberg. dm-zap: Host-based FTL for ZNS SSDs. <https://github.com/westerndigitalcorporation/dm-zap>, 2021.
- [25] Hans Holmberg. RocksDB: ZenFS Storage Backend. <https://github.com/westerndigitalcorporation/zenfs/>, 2021.
- [26] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 375–390, Santa Clara, CA, February 2017. USENIX Association.
- [27] Ping Huang, Guanying Wu, Xubin He, and Weijun Xiao. An aggressive worn-out flash block management scheme to alleviate ssd performance degradation. In *Proceedings of the Ninth European Conference on Computer Systems*, page 22. ACM, 2014.
- [28] INCITS T10 Technical Committee. Information technology - Zoned Block Commands (ZBC). Draft Standard T10/BSR INCITS 536, American National Standards Institute, Inc., September 2014. Available from <http://www.t10.org/>.
- [29] INCITS T13 Technical Committee. Information technology - Zoned Device ATA Command Set (ZAC). Draft Standard T13/BSR INCITS 537, American National Standards Institute, Inc., December 2015. Available from <http://www.t13.org/>.
- [30] William K Josephson, Lars A Bongo, Kai Li, and David Flynn. DFS: A file system for virtualized flash storage. *ACM Transactions on Storage (TOS)*, 6(3):14, 2010.
- [31] Dawoon Jung, Jeong-UK Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. Superblock FTL: A Superblock-Based Flash Translation Layer with a Hybrid Address Translation Scheme. *ACM Transactions on Embedded Computing Systems*, 9(4):1–41, March 2010.
- [32] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed Solid-state Drive. In *6th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.
- [33] Jaeho Kim, Donghee Lee, and Sam H. Noh. Towards SLO Complying SSDs Through OPS Isolation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 183–189, Santa Clara, CA, 2015. USENIX Association.
- [34] Jesung Kim, Jong Min Kim, S.H. Noh, Sang Lyul Min, and Yookun Cho. A Space-efficient Flash Translation Layer for CompactFlash Systems. *Consumer Electronics, IEEE Transactions on*, 48(2):366–375, May 2002.
- [35] D. Le Moal, Z. Bandic, and C. Guyot. Shingled File System Host-side Management of Shingled Magnetic Recording Disks. In *2012 IEEE International Conference on Consumer Electronics (ICCE)*, pages 425–426, 2012.
- [36] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, 2015.
- [37] S. Lee, K. Fleming, J. Park, Ha K, Adrian Caulfield, Steven Swanson, Arvind, and J. Kim. BlueSSD: An Open Platform for Cross-layer Experiments for NAND Flash-based SSDs. In *Proceedings of the 2010 Workshop on Architectural Research Prototyping*, January 2010.
- [38] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-managed flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 339–353, Santa Clara, CA, February 2016. USENIX Association.
- [39] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S Gunawi. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator. In *16th USENIX Conference on File and Storage Technologies (FAST)*, pages 83–90, 2018.
- [40] Wojciech Malikowski. SPDK Open-Channel SSD FTL. <https://spdk.io/doc/ftl.html>, 2018.
- [41] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, 2nd edition, 2014.
- [42] Damien Le Moal. f2fs: Zoned block device support. <https://www.spinics.net/lists/linux-fsdevel/msg103443.html>, 2014.
- [43] Damien Le Moal. f2fs: Zoned block device support. <https://www.spinics.net/lists/linux-fsdevel/msg103443.html>, 2016.
- [44] Damien Le Moal. dm-zoned: Zoned Block Device device mapper. <https://lwn.net/Articles/714387/>, 2017.

- [45] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. The Log-structured Merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, June 1996.
- [46] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’14*, page 471–484, New York, NY, USA, 2014. Association for Computing Machinery.
- [47] Adrian Palmer. SMRFFS-EXT4—SMR Friendly File System. https://github.com/Seagate/SMR_FS-EXT4, 2015.
- [48] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. A Reconfigurable FTL (Flash Translation Layer) Architecture for NAND Flash-based Applications. *ACM Trans. Embed. Comput. Syst.*, 7(4):38:1–38:23, August 2008.
- [49] Rekha Pitchumani, James Hughes, and Ethan L. Miller. SMRDB: Key-Value Data Store for Shingled Magnetic Recording Disks. In *Proceedings of the 8th ACM International Systems and Storage Conference, SYSTOR ’15*, New York, NY, USA, 2015. Association for Computing Machinery.
- [50] Aravind Ramesh, Hans Holmberg, and Shin’ichiro Kawasaki. fio zone capacity support. <https://www.spinics.net/lists/fio/msg08752.html>, 2020.
- [51] Aravind Ramesh, Damien Le Moal, and Niklas Cassel. f2fs: Zoned Namespace support. <https://www.mail-archive.com/linux-f2fs-devel@lists.sourceforge.net/msg17567.html/>, 2020.
- [52] Hannes Reinecke. Support for zoned block devices. <https://lwn.net/Articles/694966/>, July 2016.
- [53] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.
- [54] Mohit Saxena, Yiying Zhang, Michael M Swift, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Getting Real: Lessons in Transitioning Research Simulations into Hardware Systems. In *11th USENIX Conference on File and Storage Technologies (FAST)*, pages 215–228, 2013.
- [55] Seastar. Ceph SeaStore optimizes for NVMe SSDs. <https://docs.ceph.com/en/latest/dev/seastore/>, March 2020.
- [56] Samsung Semiconductor. Performance Benefits of Running RocksDB on Samsung NVMe SSDs, 2015.
- [57] Ryan Smith. SSD Cost Pricing Calculator. <https://www.soothsawyer.com/ssd-cost-pricing-calculator/>, 2019.
- [58] Yong Ho Song, Sanghyuk Jung, Sang-Won Lee, and Jin-Soo Kim. Cosmos openSSD: A PCIe-based open source SSD platform. *Proc. Flash Memory Summit*, 2014.
- [59] Theano Stavrinos, Daniel Berger, Ethan Katz-Bassett, and Wyatt Lloyd. Don’t Be a Blockhead: Zoned Namespaces Make Work on Conventional SSDs Obsolete. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS XVIII)*. ACM, 2021.
- [60] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-tree based Key-value Store on Open-Channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*, page 16. ACM, 2014.
- [61] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-Tree Based Key-Value Store on Open-Channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems, EuroSys ’14*, New York, NY, USA, 2014. Association for Computing Machinery.
- [62] Yeong-Jae Woo and Jin-Soo Kim. Diversifying Wear Index for MLC NAND Flash Memory to Extend the Lifetime of SSDs. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, page 6. IEEE Press, 2013.
- [63] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. Tiny-Tail Flash: Near-perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. *ACM Transactions on Storage (TOS)*, 13(3):22, 2017.
- [64] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don’t Stack Your Log on My Log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, 2014.
- [65] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. GearDB: A GC-free Key-Value Store on HM-SMR Drives with Gear Compaction. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 159–171, Boston, MA, February 2019. USENIX Association.

- [66] Jinsoo Yoo, Youjip Won, Joongwoo Hwang, Sooyong Kang, Jongmoo Choil, Sungroh Yoon, and Jaehyuk Cha. VSSIM: Virtual Machine-based SSD Simulator. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2013.
- [67] Yiyang Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, Berkeley, CA, USA, 2012.

MapperX: Adaptive Metadata Maintenance for Fast Crash Recovery of DM-Cache Based Hybrid Storage Devices

Lujia Yin
ylj1992nudt@gmail.com
NUDT

Li Wang
laurence.liwang@gmail.com
Didi Chuxing

Yiming Zhang
zym@nicexlab.com (Corresponding)
NiceX Lab, NUDT

Yuxing Peng
pengyuxing@aliyun.com
NUDT

Abstract

DM-cache is a component of the device mapper of Linux kernel, which has been widely used to map SSDs and HDDs onto higher-level virtual block devices that take fast SSDs as a cache for slow HDDs to achieve high I/O performance at low monetary cost. While enjoying the benefit of persistent caching where SSDs accelerate normal I/O without affecting durability, the current design of DM-cache suffers from long crash recovery times (at the scale of hours) and low availability. This is because its metadata of dirty bits has to be *asynchronously* persisted for high I/O performance, which consequently causes *all* cached data on SSDs to be assumed dirty and to be recovered after the system is restarted.

This paper presents MapperX, a novel extension to DM-cache that uses an on-disk adaptive bit-tree (ABT) to *synchronously* maintain the metadata of dirty bits in a hierarchical manner. Leveraging spatial locality of block writes, MapperX achieves controlled metadata persistence overhead with fast crash recovery by adaptively adding/deleting leaves in the ABT where different levels represent the states of blocks with different granularity. We have implemented MapperX for Linux DM-cache module. Experimental results show that the MapperX based hybrid storage device outperforms the original DM-cache based hybrid device by orders of magnitude in crash recovery times while only introducing negligible metadata persistence overhead.

1 Introduction

SSDs (solid state drives) are preferable to HDDs (hard disk drives) in building cloud storage systems [11, 31, 33, 35, 43, 44, 51, 65, 66] as SSDs significantly outperform HDDs in random small I/O [13, 14, 26, 27, 40] which is dominant in the cloud [44, 48]. Since currently SSDs are still much more expensive than HDDs, in recent years we see a trend of adopting HDD-SSD hybrid storage for high I/O performance at low monetary cost. For instance, URSA [38] is a distributed block storage system which stores primary

replicas on SSDs and replicates backup replicas on HDDs; and SSHD [28] integrates a small SSD inside a large HDD of which the SSD acts as a cache.

As the demand of HDD-SSD hybrid storage increases, Linux kernel has supported users to combine HDDs and SSDs to jointly provide virtual block storage service. DM-cache [5] is a component of the device mapper [4] in the kernel, which has been widely used in industry to map SSDs and HDDs onto higher-level virtual block devices that take fast SSDs as a cache for slow HDDs. DM-cache records the mapping between SSDs and HDDs for each cached block in its metadata. When DM-cache adopts the default writeback mode, a block write will go only to the SSD cache and get acknowledged after being marked dirty, so that the dirty block could be *demoted* from the SSD cache to the HDD later in a batch for accelerating random small I/O. Linux kernel also provides other modules (Bcache [3] and Flashcache [9]) which have similar functionalities with DM-cache.

While enjoying the performance benefit of persistent caching without affecting data durability [68], in the current design of DM-cache and its variations the metadata of dirty bits has to be *asynchronously* persisted (with a hard-coded period of one second), otherwise the synchronous update overhead of the metadata for each write would be overwhelming. Unfortunately, asynchronous metadata update causes *all* cached data on SSDs to be assumed dirty once the system crashes and gets restarted, which results in long crash recovery times and consequently low availability. For example, in our production storage cluster we use DM-cache to combine SSDs with HDDs for hybrid block storage, and it will take more than two hours (depending on the locality of workloads) to recover from a crash even if most cached blocks are clean. The low availability caused by asynchronous metadata update prevents Linux kernel's hybrid storage mechanisms from being widely applied in availability-sensitive scenarios.

To address this problem, in this paper we present MapperX, a novel extension to DM-cache that uses an on-

disk adaptive bit-tree (ABT) to *synchronously* maintain the metadata of dirty bits in a hierarchical manner. Workloads in the cloud usually have adequate write locality [36, 38, 48], which can be exploited to use one bit to represent the state of a range of consecutive blocks and thus effectively reduce the number of actual persistence operations for synchronous metadata update. Leveraging spatial locality of block writes, MapperX achieves controlled metadata persistence overhead with fast crash recovery by adaptively adding/deleting leaves at different levels in the ABT, which represent the states of blocks with different granularity.

We have implemented MapperX for Linux DM-cache module. Experimental results show that for workloads with certain localities the MapperX based hybrid storage device outperforms the original DM-cache based hybrid device by orders of magnitude in crash recovery times while only introducing negligible metadata persistence overhead.

The rest of this paper is organized as follows. Section 2 introduces the background and problem of DM-cache. Section 3 presents the design of MapperX. Section 4 evaluates the performance of MapperX and compares it with the original DM-cache. Section 5 discusses related work. And finally Section 6 concludes the paper.

2 Background

2.1 DM-Cache Overview

DM-cache is a component of the Linux kernel’s device mapper, a volume management framework that allows various mappings to be created between physical and virtual block devices. DM-cache allows one or more fast flash-based SSDs (cache devices) to act as a cache for one or more slower mechanical HDDs (origin devices). In this section we briefly introduce the basic caching mechanism of DM-cache.

Like most cache solutions, DM-cache has three operating modes, namely, *writeback*, *writethrough*, and *passthrough*, among which only the default *writeback* mode can accelerate small writes (by asynchronously persisting SSD-cached data to HDDs). In this paper we focus on DM-cache’s *writeback* mode. Linux provides DM-cache with various (plug-in) cache policy modules, such as multi-queue (MQ) and stochastic multi-queue (SMQ), to determine which blocks (and when) should be migrated from an HDD to an SSD (a.k.a. *promoted*) or from an SSD to an HDD (a.k.a. *demoted*). The cache policy is orthogonal to this paper, and we simply adopt the default SMQ policy which performs the best for most workloads.

DM-cache can use either the SSD cache device or a separate metadata device to store its metadata, which includes the mapping (between the cached blocks on SSDs and the original blocks on HDDs) and the dirty bits, as well as other policy-related metadata such as per-block hit counts. DM-cache adopts a fixed (but configurable before cache creation)

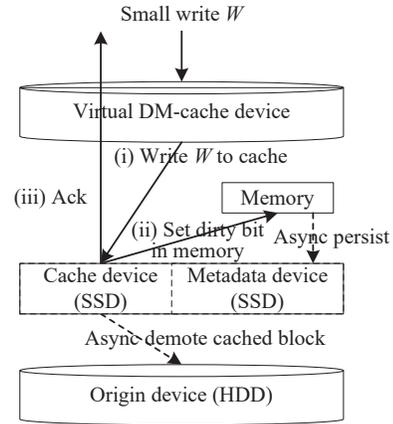


Figure 1: DM-cache maps HDDs and SSDs onto higher level virtual block devices.

block size typically between 32KB and 1MB.

Consider a small write to a virtual DM-cache device in the *writeback* mode.

If the target block is already in the SSD cache, then as shown in Fig. 1, (i) the new data (W) is written to the SSD, (ii) the corresponding bit of the block is set dirty in memory, and (iii) the write is acknowledged. The cached block will be asynchronously persisted to the HDD according to the cache policy. There are two kinds of metadata: the metadata for the cached block’s mapping (between SSD and HDD) already exists and thus needs no persistence; and the metadata for the block’s dirty bit has to be *asynchronously* persisted (one persistence per second by default), because once being synchronously persisted this update will be on the critical path of writing W which would greatly affect the performance of cached writes, as demonstrated in the next subsection. In original DM-cache the persistence of dirty bits is not for crash recovery but for (e.g., battery-backed) graceful shutdown which happens after dirty bits are (incrementally) persisted, so that up-to-date dirty-bit metadata can be read after reboot.

If the target block is not yet in the cache, then the processing is slightly complex: when the write is not aligned with a block, the block needs to be first promoted to the cache with the first kind of (mapping) metadata being persisted, after which the processing is the same as cached block writes. Note that the mapping metadata must be synchronously updated, in which case the synchronous update of the dirty-bit metadata is less harmful because the number of metadata writes only increases from one to two. In contrast, if the mapping metadata needs no update then the number of metadata writes *sharply* increases from zero to one. Mapping metadata update is largely decided by the locality of the workloads together with the replacement policy. Generally speaking, higher locality of writes causes less changes of mapping metadata which makes synchronous dirty-bit per-

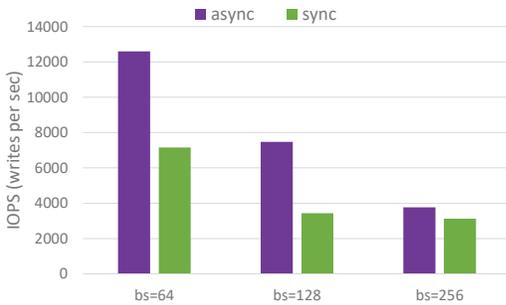


Figure 2: Synchronous updates of dirty bits severely affect the I/O performance, evaluated using one HDD + one SSD.

sistence have higher negative impact on the performance of cached writes, and vice versa.

2.2 The Main Drawback of DM-Cache

To demonstrate the runtime performance problem of persisting dirty bit for each write, we compare the IOPS of DM-cache with synchronous and asynchronous updates, respectively. We use `fiio` to perform random writes (`rw = randwrite`) and evaluate the IOPS (writes per second) with `iodepth=16`, using various cache block sizes ranging from 64 KB to 256 KB. The `fiio` write sizes are the same as the catch block sizes. The result (Fig. 2) shows that the performance is severely affected when adopting synchronous update for dirty-bit metadata, causing several times IOPS degradation. The high overhead prevents synchronous update of dirty-bit metadata from being adopted by DM-cache.

Unfortunately, asynchronous metadata update causes *all* cached data on SSDs to be assumed dirty once the system crashes and gets restarted, which results in long crash recovery times and consequently low availability. For example, in our production block storage system where we use DM-cache to combine multiple SSDs with multiple HDDs on each storage machine, it takes more than two hours (depending on the workloads) to recover from a power failure (by demoting all cached blocks) even if most blocks are clean. The low availability (caused by asynchronous metadata update) prevents Linux’s HDD-SSD hybrid cache mechanisms (like DM-cache) from being widely applied in availability-sensitive scenarios.

3 MapperX Design

3.1 Synchronous Metadata Update

The timing of dirty-bit metadata update is a dilemma for HDD-SSD hybrid devices. The asynchronous update mechanism periodically updates dirty bits for not affecting normal writes but suffers from long crash recovery time (since all SSD-cached blocks have to be assumed dirty and get recovered even if most of them are clean); while the

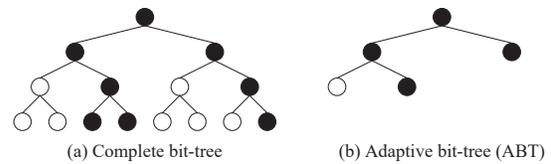


Figure 3: MapperX maintains an on-disk adaptive bit-tree (ABT), a summary of the complete bit-tree (CBT) of which the leaves represent the states of all HDD blocks. Black nodes represent dirty = `true`. ABT is *synchronously* updated for each write, but most updates do not need persistence to SSD since one bit represents a set of consecutive blocks.

synchronous update mechanism keeps all dirty bits up-to-date for fast crash recovery but greatly increases the latency of cached writes. Although the-state-of-the-art caching solution (DM-cache) adopts asynchronous update for high I/O performance, in this paper we propose to take the synchronous update mechanism for fast crash recovery.

The challenge is that there is no trade-offs for the timing of dirty-bit metadata update, because once the metadata is asynchronously updated even only one outdated bit state would require demotion of all cached blocks to ensure data durability after crash recovery. To address this challenge, our key idea is to adjust the granularity, instead of the timing, of dirty-bit metadata update to smoothly trade off between normal I/O performance and crash recovery time.

Workloads in the cloud usually have adequate write locality which can be leveraged to use one bit to represent the state of a range of consecutive blocks, as long as we can (roughly) know the effective range of the bit. Note that false positives of the effective range are not critical: if one clean block is wrongly included in the effective range of a dirty bit, the price is simply an unnecessary demotion of that block in crash recovery.

There is an in-memory bitmap that precisely records the state of every block. We first extend the in-memory bitmap to a (logical) hierarchical complete bit-tree or CBT (as shown in Fig. 3(left)), where the leaves are the bits of the bitmap and each inner node is the disjunction of its direct children. Then, MapperX maintains a *summary* of the CBT on the metadata device, which we refer to as on-disk *adaptive bit-tree* (ABT), as shown in Fig. 3(right). The ABT is synchronously updated for each write request, but most updates do not cause disk writes for metadata persistence, as discussed below. Each inner node of the ABT represents a range of consecutive blocks, and thus only the first dirty block within the range causes a write to the metadata device and subsequent writes of blocks in the range need no persistence. Initially, the ABT only has a root node representing the states of *all* cache blocks, which will change from clean to dirty after the first block write.

MapperX proposes a synchronous ABT update mecha-

Algorithm 1 Synchronous bit-tree update of MapperX

```
1: procedure BITTREEUPDATE(Block  $b$ , Adaptive bit-  
   tree  $abt$ )  
2:   Update in-memory bitmap by  $b$   
3:   Calculate affected inner nodes of complete bit-tree  
   ( $cbt$ ) ▷ Dirty if any child dirty, clean if all children clean  
4:   if  $b$  causes leaf node  $L$  of  $abt$  to become dirty then  
5:     Update  $abt$  on disk according to  $cbt$   
6:   end if  
7:   return SUCCESS  
8: end procedure  
  
9: procedure PERIODICADJUST(Period  $p$ , SLA  $n$ , Adap-  
   tive bit-tree  $abt$ )  
10:   $W \leftarrow$  total number of client writes during  $p$   
11:   $N \leftarrow$  total number of metadata writes during  $p$   
12:  if  $N/W \geq 1/10^n$  then ▷ Too many metadata writes  
13:     $parents \leftarrow$  all direct parents of the leaves in  $abt$   
14:     $target\_parent \leftarrow$  the parent from  $parents$  which  
    has experienced the most metadata writes during  $p$   
15:    Delete all children of  $target\_parent$  in  $abt$   
16:  else  
17:     $target\_leaf \leftarrow$  the leaf from all leaf nodes of  $abt$   
    which has experienced the least metadata writes in  $p$   
18:    Generate  $d$  children for  $target\_leaf$  in  $abt$  and  
    set their states according to  $cbt$  ▷  $d$  is the degree of  $abt$   
19:  end if  
20:  return SUCCESS  
21: end procedure
```

nism (Algorithm 1), which can adaptively adjust the metadata persistence frequency. The basic idea is to control the effective range of the corresponding bits of the leaves (i.e., the granularity of metadata update) by adding/deleting leaves in the ABT. Leaves at higher levels (farther from the root) in the ABT represent the states of a smaller range of blocks and thus increasing the levels of the ABT will increase the metadata persistence overhead while reducing the expected recovery time, and vice versa. When adding child leaves to an existing leaf node, the information about which children are dirty can be obtained from the logical CBT (calculated from the in-memory bitmap).

Since the user-perceived I/O performance is usually described as tail latencies, e.g., 99.9th percentile latency guarantee requires only one out of 1000 writes can be affected by dirty-bit metadata update, we use the number of nines (n) of the SLA (service-level agreement) to control the summary levels of the ABT.

The BITTREEUPDATE procedure first updates the in-memory bitmap and calculates the affected nodes in the complete bit-tree (Lines 2~3). Then, if the current block write causes a leaf node to change from clean to dirty, we will persist the updated ABT (Lines 4~6).

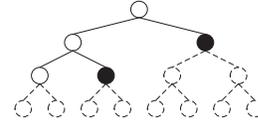


Figure 4: Example of a virtual ABT (degree $d = 2$, level $m = 4$) stored in a flat bit array for the actual ABT (Fig. 3(left)). For $d^{m-1} = 8$ leaves (representing 8 blocks), there are totally $\sum_{i=0}^{m-1} d^i = \frac{d^m-1}{d-1} = 15$ nodes in the tree, which can be stored using 15 bits (0010 1000 0000 000).

The PERIODICADJUST procedure decides whether to add or delete leaf nodes in the ABT according to the statistics of the last period (p). The period is configurable and by default set as one second. If there are too many metadata writes compared to the SLA in p , then we will remove all the children of the parent that has experienced the most metadata writes during p (Lines 12~15). Otherwise we will add children to the leaf node that has experienced the least metadata writes during p (Lines 16~18). Similar to the original DM-cache, MapperX can synchronously or asynchronously update the ABT (without add/deleting leaves) when the dirty blocks are demoted to the HDD.

3.2 Fast Crash Recovery

Since the ABT is synchronously updated for every write request, the leaf nodes in the ABT has no false negatives for dirty states. That is, when a leaf is not dirty, each of the blocks it represents are guaranteed to be not dirty and we can safely skip these blocks in crash recovery. Therefore, the recovery procedure of MapperX-based DM-cache devices is straightforward: for each dirty leaf L of the ABT, demote all SSD-cached blocks of L to the HDD.

3.3 Implementation

We have implemented MapperX on CentOS 7 by augmenting the original DM-cache with BITTREEUPDATE and PERIODICADJUST in Algorithm 1, and realizing the in-memory CBT and the on-disk ABT structures in Fig. 3.

In order not to introduce extra storage overhead, we reuse DM-cache's four-byte dirty-bit metadata structure of each cached block, of which DM-cache uses only the last two bits (a dirty bit and a valid bit) leaving the first 30 bits available for MapperX. We organize the first 30 bits of all cached blocks' metadata structures into a flat bit array. To minimize the update overhead of adding/deleting leaves, we store ABT as the virtual ABT or V-ABT (Fig. 4) which has the same numbers of levels and leaves as the CBT but where only the ABT's *dirty leaves* are 1 and all other inner/leaf nodes are 0. We use the flat bit array for statically representing the states of all inner/leaf nodes in the V-ABT (each bit for one node started from the root in a breadth-first manner).

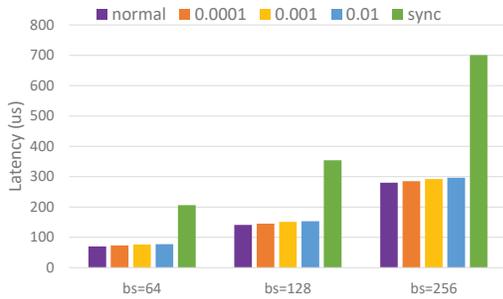


Figure 5: MapperX ($\beta = 0.01, 0.001, 0.0001$) vs. DM-cache (normal) in mean latency.

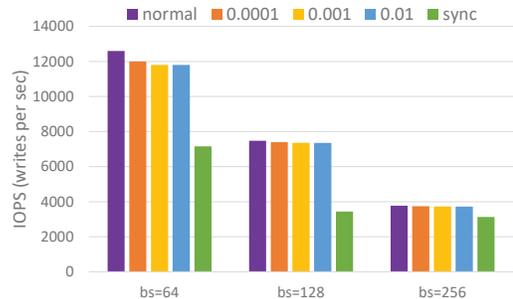


Figure 7: MapperX ($\beta = 0.01, 0.001, 0.0001$) vs. DM-cache (normal) in IOPS (writes per sec).

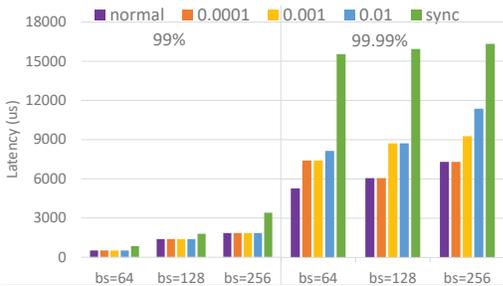


Figure 6: MapperX ($\beta = 0.01, 0.001, 0.0001$) vs. DM-cache (normal) in tail latency.

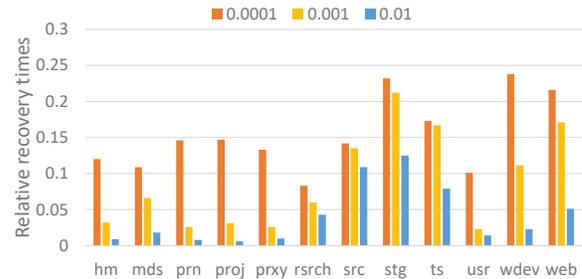


Figure 8: Recovery times of MapperX ($\beta = 0.01, 0.001, 0.0001$) relative to DM-cache for MSR traces.

4 Evaluation

Our test machine has an Intel gold 6240 36-core 2.60GHz CPU and 64GB RAM, running CentOS 7. The machine has one SATA 7200RPM 2TB HDD and one NVMe 400GB SSD. We configure the DM-cache virtual block device with 1TB HDD storage device, 128GB SSD cache device, and 1GB SSD metadata device. The client runs the `fiio` benchmark tool [8] to perform random writes (`rw=randwrite`) that all hit the SSD cache for three hundred seconds. Note that cache miss should be avoided in this test because otherwise the poor performance of HDD storage would dominate the overall performance making MapperX and DM-cache have no difference.

4.1 Micro Benchmarks

We first compare the latency of random writes of MapperX and DM-cache. The cache block size (bs) is 64KB \sim 256KB, and the `fiio` write block size is equal to the cache block size. The degree of the tree is $d = 4$. We evaluate the latency using one `fiio` thread with `iodepth = 1`. We use β to represent the expected ratio of metadata writes to all writes (i.e., $\beta = 1/10^n$ where n is the SLA) and set $\beta = 0.01, 0.001, 0.0001$, respectively. We set `max_level = 7` (maximum number of levels), which limits the ABT to have $d^{\text{max_level}} = 16K$ leaves each representing 1024, 512, and 256 blocks for $bs = 64\text{KB}, 128\text{KB},$ and 256KB (for the 1TB HDD storage),

respectively. The original DM-cache adopts asynchronous metadata update (which equals to set `max_level = 1`). The results are shown in Fig. 5 and Fig. 6 respectively for the mean and tail latencies. The results show that the latency overhead of MapperX slightly increases as β increases from 0.0001 to 0.01, but the overhead is small compared to the original DM-cache.

We compare the IOPS (number of writes per second) of random writes of MapperX and DM-cache. The configuration is the same as that in the latency test except that we use one `fiio` thread with `iodepth = 16`. The result is shown in Fig. 7, where MapperX has similar IOPS with DM-cache, which proves that the IOPS overhead of MapperX is small compared to the original DM-cache. Note that higher latency does not necessarily cause lower throughput (and vice versa), because NVMe SSD supports high parallelism which enables it to mask I/O delays with parallel I/O requests flying over the wire and waiting in the pipeline.

4.2 Trace-Driven Evaluation

We compare the recovery performance of MapperX ($bs = 128$) and the original DM-cache, using the public I/O traces from Microsoft Research [1] that capture block-level I/O (below the filesystem cache) of various desktop/server applications running for one week. The result is shown in Fig. 8, where the original DM-cache has to recover all the cached blocks because of its asynchronous dirty-bit metadata

update. In contrast, MapperX only needs to recover much fewer blocks owing to its synchronous dirty-bit metadata update. For example, the recovery time of MapperX with $\beta = 0.01$ for the proj trace is only 0.6% that of DM-cache. Generally speaking, less locality leads to fewer ABT levels, fewer normal-time ABT updates, and longer crash recovery time (due to higher false-positive rates), with the design of the original DM-cache at the extreme end.

5 Related Work & Discussion

DM-cache [5], Bcache [3] and Flashcache [9] are Linux kernel modules which are used to combine fast SSDs with slow HDDs as a virtual block device. The difference is that Bcache utilizes a btree cache structure, while Flashcache's cache is structured as a set-associative hash table. LVM-cache [12] is built on top of DM-cache using logical volumes to avoid calculation of block offsets and sizes. DM-cache can also be used as the client-side local storage for caching of virtual machines in storage area networks (SANs).

DM-cache and its variations asynchronously update dirty bits and thus cannot recover from crashes with up-to-date dirty-bit information. Consequently, all cached blocks on SSD have to be assumed dirty, which makes cached blocks have to be written to HDD. In contrast, MapperX uses ABT to synchronously update dirty bits, providing flexibility of whether to write dirty data to HDD on recovery. Since recovery from crashes takes time, it is natural for MapperX to take a little more time (usually several minutes depending on the volume of dirty data) for demotion.

In addition to DM-cache/LVM-cache/Bcache/Flashcache provided by Linux kernel, several SSD-HDD hybrid designs use SSD as a cache layer. For example, Nitro [37] designs a capacity-optimized SSD cache for primary storage. Solid State Hybrid Drives (SSHD [28]) integrate an SSD inside a traditional HDD and realize SSD cache in a way similar to Nitro. URSA [38] is a distributed block storage system which stores primary replicas on SSDs and replicates backup replicas on HDDs. Griffin [60] designs a hybrid storage device that uses HDDs as a write cache for SSDs to extend SSD lifetimes. Compared to these studies, MapperX mainly focuses on the tradeoff between normal write performance and recovery times, using the ABT to adaptively adjust the range represented by the leaves.

Journal [38] based metadata write is inefficient [63] for DM-cache because of expensive write ordering [22]. For consistency, journal-based solutions always impose ordering constraint on writes (e.g., data \rightarrow sync \rightarrow metadata \rightarrow sync) [21], which both increases latency and decreases throughput. In DM-cache and MapperX, if the blocks are already in the cache then the writes need not update the mapping metadata, so they only need one SSD write for DM-cache (Fig. 1) and (if ABT unchanged) for MapperX.

Log-structured solutions [24] are inefficient for DM-

cache, because log-style writing always causes changes of the SSD-HDD mapping metadata (due to new block allocation) which has to be updated even for cache-hit write requests. In contrast, this can be avoided by DM-cache and MapperX for cache-hit write requests when mapping metadata keeps unchanged, where only one metadata write is needed (Fig. 1). Moreover, garbage collection (GC) [49] for old versions is expensive for log-structured systems.

Hardware-based, out-of-band solutions [17] require special hardware and driver supports with high programming complexity, and recently-emerging open-channel SSDs [7] are not readily available. As far as we know, large cloud providers that own millions of SSDs only have a few thousands SSDs with open-channel customization support. Further, synchronous update of SSD page states can be viewed as a special case of ABT where each leaf represents a page. This is inefficient for workloads with good locality where page states frequently change: writing a page back to HDD requires to synchronously change its state (clean) on SSD, which can be avoided by ABT if the higher-level node state keeps unchanged (dirty). On the other hand, NVM (non-volatile memory) [6] based solutions [15, 19, 20, 62, 71] are promising but expensive. NVM is still uncommon in the cloud, and the relatively-small NVM is expected to be used in more critical scenarios.

6 Conclusion

This paper presents MapperX, a novel extension to DM-cache that uses an on-disk bit-tree to *synchronously* persist the dirty-bit metadata in a hierarchical manner. Experimental results show that MapperX significantly outperforms DM-cache in crash recovery times while only introducing negligible metadata write overhead. In our future work, we will apply EC [34, 39, 45, 55, 70] for ABT, study the impact of MapperX on durability and consistency [30, 32, 46, 52, 58], apply ABT in distributed file systems [21, 23, 29, 36, 50, 53, 56, 59] and object storage systems [2, 16, 47, 64, 67], apply bloom filters [25, 42, 57, 61, 69] and compression [18, 41, 54] in ABT, and improve ABT's adjustment policy.

The source code of MapperX is available at [10].

Acknowledgement

We would like to thank Amy Tai, our shepherd, and the anonymous reviewers for their insightful comments. We thank Shun Gai for helping in the experiments, and we thank the Didi Cloud Storage Team for their discussion. Lujia Yin and Li Wang are co-primary authors. Lujia Yin implemented some parts of MapperX when he was an intern at Didi Chuxing. This research is supported by the National Key R&D Program of China (2018YFB2101102) and the National Natural Science Foundation of China (NSFC 61772541, 61872376 and 61932001).

References

- [1] <http://iotta.snia.org/traces/388>.
- [2] <https://aws.amazon.com/s3/>.
- [3] <https://bcache.evildpiepirate.org/>.
- [4] https://en.wikipedia.org/wiki/Device_mapper.
- [5] <https://en.wikipedia.org/wiki/Dm-cache>.
- [6] https://en.wikipedia.org/wiki/Non-volatile_memory.
- [7] https://en.wikipedia.org/wiki/Open-channel_SSD.
- [8] <https://fio.readthedocs.io/en/latest/>.
- [9] <https://github.com/facebookarchive/flashcache>.
- [10] <https://github.com/nicexlab/mapperx>.
- [11] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
- [12] <https://man7.org/linux/man-pages/man7/lvmcache.7.html>.
- [13] ANAND, A., MUTHUKRISHNAN, C., KAPPES, S., AKELLA, A., AND NATH, S. Cheap and large cams for high performance data-intensive networked systems. In *NSDI* (2010), USENIX Association, pp. 433–448.
- [14] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. Fawn: a fast array of wimpy nodes. In *SOSP* (2009), J. N. Matthews and T. E. Anderson, Eds., ACM, pp. 1–14.
- [15] ANDERSON, T. E., CANINI, M., KIM, J., KOSTIĆ, D., KWON, Y., PETER, S., REDA, W., SCHUH, H. N., AND WITCHEL, E. Assise: Performance and availability via client-local NVM in a distributed file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 1011–1027.
- [16] BEAVER, D., KUMAR, S., LI, H. C., SOBEL, J., AND VAJGEL, P. Finding a needle in haystack: Facebook’s photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI’10, USENIX Association, pp. 47–60.
- [17] BJØRLING, M., GONZALEZ, J., AND BONNET, P. Lightnvm: The linux open-channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (Santa Clara, CA, Feb. 2017), USENIX Association, pp. 359–374.
- [18] BORSTNIK, U., VANDEVONDELE, J., WEBER, V., AND HUTTER, J. Sparse matrix multiplication: The distributed block-compressed sparse row library. *Parallel Comput.* 40, 5-6 (2014), 47–58.
- [19] CHEN, Y., LU, Y., CHEN, P., AND SHU, J. Efficient and consistent NVMM cache for ssd-based file system. *IEEE Trans. Computers* 68, 8 (2019), 1147–1158.
- [20] CHENG, W., LI, C., ZENG, L., QIAN, Y., LI, X., AND BRINKMANN, A. Nvmm-oriented hierarchical persistent client caching for lustre. *ACM Trans. Storage* 17, 1 (2021), 6:1–6:22.
- [21] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 228–243.
- [22] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency without ordering. In *Usenix Conference on File and Storage Technologies* (2012).
- [23] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 133–146.
- [24] DAI, Y., XU, Y., GANESAN, A., ALAGAPPAN, R., KROTH, B., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. From wiskey to bourbon: A learned index for log-structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 155–171.
- [25] DAI, Z., AND SHRIVASTAVA, A. Adaptive learned bloom filter (ada-bf): Efficient utilization of the classifier with application to real-time information filtering on the web. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual* (2020), H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds.
- [26] DEBNATH, B., SENGUPTA, S., AND LI, J. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2011), SIGMOD ’11, ACM, pp. 25–36.
- [27] DEBNATH, B. K., SENGUPTA, S., AND LI, J. Flashstore: High throughput persistent key-value store. *PVLDB* 3, 2 (2010), 1414–1425.
- [28] DORDEVIC, B., TIMCENKO, V., AND RAKAS, S. B. Sshd: Modeling and performance analysis. *INFOTEH-JAHORINA* 15, 3 (2016), 526–529.
- [29] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *SOSP* (2003), pp. 29–43.
- [30] GRAY, C., AND CHERITON, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1989), SOSP ’89, ACM, pp. 202–210.
- [31] HARTER, T., BORTHAKUR, D., DONG, S., AIYER, A., TANG, L., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis of hdfs under hbase: A facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (2014), pp. 199–212.
- [32] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492.
- [33] HILDEBRAND, D., AND HONEYMAN, P. Exporting storage systems in a scalable manner with pnfs. In *22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST’05)* (2005), IEEE, pp. 18–27.
- [34] KOSAIAN, J., RASHMI, K. V., AND VENKATARAMAN, S. Parity models: erasure-coded resilience for prediction serving systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019* (2019), T. Brecht and C. Williamson, Eds., ACM, pp. 30–46.
- [35] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *ACM SIGPLAN Notices* (1996), vol. 31, ACM, pp. 84–92.
- [36] LEUNG, A. W., PASUPATHY, S., GOODSON, G. R., AND MILLER, E. L. Measurement and analysis of large-scale network file system workloads. In *USENIX annual technical conference* (2008), vol. 1, pp. 2–5.
- [37] LI, C., SHILANE, P., DOUGLIS, F., SHIM, H., SMALDONE, S., AND WALLACE, G. Nitro: A capacity-optimized ssd cache for primary storage. In *USENIX Annual Technical Conference* (2014), pp. 501–512.
- [38] LI, H., ZHANG, Y., LI, D., ZHANG, Z., LIU, S., HUANG, P., QIN, Z., CHEN, K., AND XIONG, Y. Ursa: Hybrid block storage for cloud-scale virtual disks. In *Proceedings of the Fourteenth EuroSys Conference 2019* (2019), ACM, p. 15.
- [39] LI, H., ZHANG, Y., ZHANG, Z., LIU, S., LI, D., LIU, X., AND PENG, Y. Parix: speculative partial writes in erasure-coded systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017), USENIX Association, pp. 581–587.

- [40] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 1–13.
- [41] LIU, Y., AND SCHMIDT, B. Lightspmv: Faster cuda-compatible sparse matrix-vector multiplication using compressed sparse rows. *J. Signal Process. Syst.* 90, 1 (2018), 69–86.
- [42] LU, J., WAN, Y., LI, Y., ZHANG, C., DAI, H., WANG, Y., ZHANG, G., AND LIU, B. Ultra-fast bloom filters using SIMD techniques. *IEEE Trans. Parallel Distributed Syst.* 30, 4 (2019), 953–964.
- [43] MEYER, D. T., AGGARWAL, G., CULLY, B., LEFEBVRE, G., FEELEY, M. J., HUTCHINSON, N. C., AND WARFIELD, A. Parallax: virtual disks for virtual machines. In *ACM SIGOPS Operating Systems Review* (2008), vol. 42, ACM, pp. 41–54.
- [44] MICKENS, J., NIGHTINGALE, E. B., ELSON, J., GEHRING, D., FAN, B., KADAV, A., CHIDAMBARAM, V., KHAN, O., AND NAREDDY, K. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), pp. 257–273.
- [45] MITRA, S., PANTA, R. K., RA, M., AND BAGCHI, S. Partial-parallel-repair (PPR): a distributed technique for repairing erasure coded storage. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016* (2016), C. Cadar, P. R. Pietzuch, K. Keeton, and R. Rodrigues, Eds., ACM, pp. 30:1–30:16.
- [46] MOHAN, J., MARTINEZ, A., PONNAPALLI, S., RAJU, P., AND CHIDAMBARAM, V. Finding crash-consistency bugs with bounded black-box crash testing. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018* (2018), A. C. Arpaci-Dusseau and G. Voelker, Eds., USENIX Association, pp. 33–50.
- [47] MURALIDHAR, S., LLOYD, W., ROY, S., HILL, C., LIN, E., LIU, W., PAN, S., SHANKAR, S., SIVAKUMAR, V., TANG, L., AND KUMAR, S. F4: Facebook’s warm blob storage system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI’14, USENIX Association, pp. 383–398.
- [48] NARAYANAN, D., DONNELLY, A., AND ROWSTRON, A. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)* 4, 3 (2008), 10.
- [49] NGUYEN, K., FANG, L., XU, G., DEMSKY, B., LU, S., ALAMIAN, S., AND MUTLU, O. Yak: A high-performance big-data-friendly garbage collector. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 349–365.
- [50] NIAZI, S., ISMAIL, M., HARIDI, S., DOWLING, J., GROHSSCHMIEDT, S., AND RONSTRÖM, M. Hopsfs: Scaling hierarchical file system metadata using newsq databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (2017), pp. 89–104. 00091.
- [51] NIGHTINGALE, E. B., ELSON, J., FAN, J., HOFMANN, O., HOWELL, J., , AND SUZUE, Y. Flat datacenter storage. In *OSDI* (2012).
- [52] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2014), USENIX ATC’14, USENIX Association, pp. 305–320.
- [53] PAN, S., STAVRINOS, T., ZHANG, Y., SIKARIA, A., ZAKHAROV, P., SHARMA, A., P. S. S., SHUEY, M., WAREING, R., GANGAPURAM, M., CAO, G., PRESEAU, C., SINGH, P., PATIEJUNAS, K., TIPTON, J. R., KATZ-BASSETT, E., AND LLOYD, W. Facebook’s Tectonic Filesystem: Efficiency from Exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)* (Feb. 2021), USENIX Association, pp. 217–231. 00000.
- [54] PLIGOUROUDIS, M., NUNO, R. A. G., AND KAZMIERSKI, T. Modified compressed sparse row format for accelerated fpga-based sparse matrix multiplication. In *IEEE International Symposium on Circuits and Systems, ISCAS 2020, Sevilla, Spain, October 10-21, 2020* (2020), IEEE, pp. 1–5.
- [55] RASHMI, K. V., CHOWDHURY, M., KOSAIAAN, J., STOICA, I., AND RAMCHANDRAN, K. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. 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. 401–417.
- [56] REN, K., ZHENG, Q., PATIL, S., AND GIBSON, G. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), IEEE, pp. 237–248. 00136.
- [57] SANTIAGO, L., VERONA, L. D., RANGEL, F. M., DE FARIA, F. F., MENASCHE, D. S., CAARLS, W., JR., M. B., KUNDU, S., LIMA, P. M. V., AND FRANÇA, F. M. G. Weightless neural networks as memory segmented bloom filters. *Neurocomputing* 416 (2020), 292–304.
- [58] SHI, X., PRUETT, S., DOHERTY, K., HAN, J., PETROV, D., CARRIG, J., HUGG, J., AND BRONSON, N. Flighttracker: Consistency across read-optimized online stores at facebook. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 407–423.
- [59] SHVACHKO, K. V. Hdfs scalability: The limits to growth. ; *login:: the magazine of USENIX & SAGE* 35, 2 (2010), 6–16.
- [60] SOUNDARAJAN, G., PRABHAKARAN, V., BALAKRISHNAN, M., AND WOBBER, T. Extending ssd lifetimes with disk-based write caches. In *FAST* (2010), vol. 10, pp. 101–114.
- [61] STANCIU, V. D., VAN STEEN, M., DOBRE, C., AND PETER, A. Privacy-preserving crowd-monitoring using bloom filters and homomorphic encryption. In *EdgeSys@EuroSys 2021: 4th International Workshop on Edge Systems, Analytics and Networking, Online Event, United Kingdom, April 26, 2021* (2021), A. Y. Ding and R. Mortier, Eds., ACM, pp. 37–42.
- [62] VENKATESAN, V., WEI, Q., AND TAY, Y. C. Ex-tmem: Extending transcendent memory with non-volatile memory for virtual machines. In *2014 IEEE International Conference on High Performance Computing, HPC/CSS/ICSS 2014* (2014), IEEE, pp. 966–973.
- [63] WANG, L., XUE, J., LIAO, X., WEN, Y., AND CHEN, M. LCCFS: a lightweight distributed file system for cloud computing without journaling and metadata services. *Sci. China Inf. Sci.* 62, 7 (2019), 72101:1–72101:14.
- [64] WANG, L., ZHANG, Y., XU, J., AND XUE, G. MAPX: controlled data migration in the expansion of decentralized object-based storage systems. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020* (2020), S. H. Noh and B. Welch, Eds., USENIX Association, pp. 1–11.
- [65] WANG, Y., KAPRITSOS, M., REN, Z., MAHAJAN, P., KIRUBANANDAM, J., ALVISI, L., AND DAHLIN, M. Robustness in the salus scalable block store. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013), pp. 357–370.
- [66] WARFIELD, A., ROSS, R., FRASER, K., LIMPACH, C., AND HAND, S. Parallax: Managing storage for a million machines. In *HotOS* (2005).
- [67] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), pp. 307–320.

- [68] ZHANG, Y., GUO, C., LI, D., CHU, R., WU, H., AND XIONG, Y. Cubicring: Enabling one-hop failure detection and recovery for distributed in-memory storage systems. In *NSDI* (2015), pp. 529–542.
- [69] ZHANG, Y., LI, D., CHEN, L., AND LU, X. Collaborative search in large-scale unstructured peer-to-peer networks. In *2007 International Conference on Parallel Processing (ICPP 2007), September 10-14, 2007, Xi-An, China* (2007), IEEE Computer Society, p. 7.
- [70] ZHANG, Y., LI, H., LIU, S., XU, J., AND XUE, G. PBS: an efficient erasure-coded block storage system based on speculative partial writes. *ACM Trans. Storage* 16, 1 (2020), 6:1–6:25.
- [71] ZHU, G., LU, K., WANG, X., ZHANG, Y., ZHANG, P., AND MITTAL, S. Swapx: An nvm-based hierarchical swapping framework. *IEEE Access* 5 (2017), 16383–16392.

Exploring the Design Space of Page Management for Multi-Tiered Memory Systems

Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn
Ajou University

Abstract

With the arrival of tiered memory systems comprising various types of memory, such as DRAM and SCM, the operating system support for memory management is becoming increasingly important. However, the way that operating systems currently manage pages was designed under the assumption that all the memory has the same capabilities based on DRAM. This oversimplification leads to non-optimal memory usage in tiered memory systems. This study performs an in-depth analysis of page management schemes in the current Linux design extending NUMA to support systems equipped with both DRAM and SCM (Intel’s DCPMM). In such multi-tiered memory systems, we find that the critical factor in performance is not only the *access locality* but also the *access tier* of memory. When considering both characteristics, there are several alternatives to page placement. However, current operating systems only prioritize access locality. This paper explores the design space of page management schemes, called *AutoTiering*, to use multi-tiered memory systems effectively. Our evaluation results show that our proposed techniques can significantly improve performance for various workloads, compared to the stock Linux kernel, by unlocking the potential of the multi-tiered memory hierarchy.

1 Introduction

With the advent of in-memory computing, such as data analytics, key-value stores, and graph processing, the demand for high-density DRAM has been steadily increasing in recent years [27]. However, due to the challenge of scaling DRAM density, a new class of memory has received attention to bridge the performance gap between DRAM and SSD. For example, Intel recently unveiled its non-volatile memory based on 3D Xpoint technology, called Optane DC Persistent Memory Module (DCPMM) that provides more density than DRAM while outperforming flash-based SSDs [23]. Cloud vendors such as Google, Oracle, Microsoft, and Baidu have adopted such storage class memory (SCM) in their cloud services [4, 14, 17, 25].

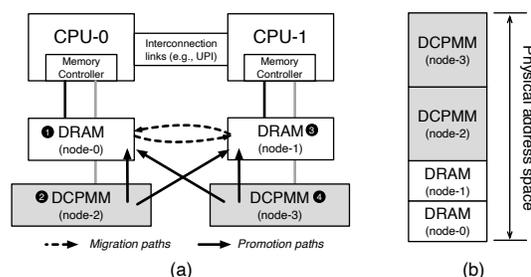


Figure 1: Software-managed tiered memory system augmented on the NUMA architecture

Since modern server systems are built with the Non-Uniform Memory Access (NUMA) architecture, future large-memory systems will take the shape of tiered memory augmented on traditional NUMA architecture, called *multi-tiered memory*. Figure 1 presents a real-world multi-tiered memory system used throughout this study. Each compute chip has two types of memory: DRAM (upper-tier) and Intel’s DCPMM (lower-tier). We configure both DRAM and DCPMM to be fully exposed to software as memory.

This paper presents that the recent advancement in Linux [15] and tiered memory studies [16, 20, 35] do not lead to optimal page placement in multi-tiered memory systems. As the new class of memory becomes part of the main memory, the critical factor in performance is not only the *access locality* but also the *access tier* of memory. However, current page placement schemes have been established for DRAM-only NUMA architecture and only consider locality between threads and memory [2, 8, 12, 13, 21, 38]. As a result, the current design is far from exploiting the potential benefits of multi-tiered memory systems. For example, suppose the local DRAM becomes full when promoting pages from the lower-tier (DCPMM) to the upper-tier (DRAM) memory. In this case, the current state of the art leaves the page on the lower-tier memory, regardless of the availability of the remote DRAM (of the upper-tier). Such a decision is reasonable for DRAM-only NUMA systems because there is no difference between alternatives. However, in multi-tiered memory sys-

tems, we cannot consider every possible alternative equivalent to the *access tier*. When placing pages, the access tier of memory should be considered before the access locality because the access tier has a more significant impact on performance.

This limitation motivates us to revisit the page management schemes of the commodity OSes and explore the design space of page management for multi-tiered memory systems. In this study, we introduce a set of new page management schemes. Our first scheme, called *AutoTiering-CPM*, conservatively looks for promotion or migration alternatives using the access tier and locality metric when failing to find the best memory node (e.g., local DRAM).

Although this conservative approach can achieve better performance by considering alternatives, such a design does not unlock the full potential of software-managed tiered memory. To effectively utilize the limited capacity of upper-tier memory, we design a page reclamation scheme tailored to multi-tiered memory systems. Our second technique is opportunistic page promotion or migration, called *AutoTiering-OPM*, which judiciously demotes pages from the upper-tier memory. To reclaim effectively, we predict the *least accessed page* as a victim in the upper-tier memory by estimating the access frequency of pages. When deciding on which page to promote, our OPM compares the page with the victim to determine which is relatively more accessed. With OPM, we can achieve better effectiveness of the upper-tier memory while reducing the memory accesses to the lower-tier memory.

Unless there is a free space in the upper-tier memory, a promotion operation waits until the completion of a demotion operation. To hide the latency of demoting pages from the critical path, we reserve a set of free pages in the upper-tier memory to serve the promotion requests immediately. When the number of reserved pages exceeds a threshold, our *kdemoted* wakes up and reclaims the least accessed page to the free page pool in the background. *kdemoted* differs from the traditional reclamation because it is only responsible for demoting pages to the lower-tier memory and not storage.

In this study, we implement our proposed schemes on top of Linux kernel v5.3. We take advantage of the AutoNUMA facility, which periodically scans memory pages and marks them inaccessible to capture non-local DRAM accesses. Once the pages are reaccessed, it incurs a page fault, called *NUMA hinting page fault*. We take the NUMA faults as demand signals for the page promotion from the DCPMM nodes or the migration from the remote DRAM node. We build the access history per page with the fault-based facility and use this information when demoting pages.

The experimental results show that our *AutoTiering* can significantly improve the performance of various applications. *GraphMat* and *graph500* show performance increases around $2.3\times$ and $6.9\times$, respectively, compared with the baseline Linux kernel [15]. Most of the SPECaccel workloads show a $2\times$ speedup. Compared to Intel’s recent approach [36], our performance improvement shows up to a $3.5\times$ speedup.

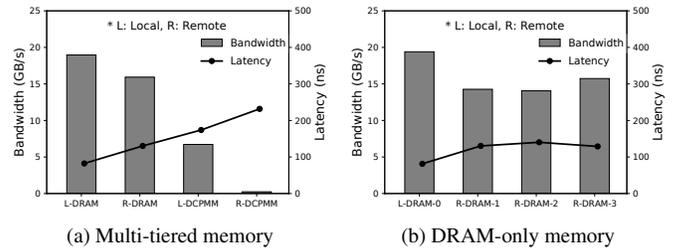


Figure 2: Memory access latency and bandwidth for multi-tiered and DRAM-only memory systems

2 Background and Motivation

2.1 Large Memory Systems

Data centers typically employ multi-chip NUMA architecture to scale up the performance of commodity servers with high core counts and memory capacity. Although this can increase the number of DIMM slots per server, scaling DRAM density is still a significant obstacle. It poses challenges in cost-effectively constructing large memory systems. Meanwhile, since SCM offers byte-addressable and non-volatile properties, it is gaining traction to bridge the performance gap between DRAM and SSD. Intel recently released the 3D XPoint non-volatile memory (DCPMM) that can be installed on DIMM without modification [23]. Many cloud vendors such as Google, Microsoft, Oracle, and Baidu have adopted Intel’s DCPMM in their cloud services [4, 14, 17, 25]. Recently, Samsung has revealed a CXL (Compute Express Link) based DRAM module attached to the system, forming tiered memory systems [1]. Since such new type of memory is not as fast as DRAM, they cannot replace DRAM entirely. Instead, future computer systems will offer a form of tiered memory architecture with DRAM and SCM.

In this study, we take advantage of DCPMM as a new tier between DRAM and SSD. Intel DCPMM provides two types of tiered memory systems that can be categorized as hardware-assisted or software-managed. In hardware-assisted mode, DCPMM is exposed to software as the main memory while DRAM acts as a hardware-managed cache, non-visible to the software. The memory controller automatically places frequently accessed data on the DRAM cache, while the rest of the data is kept on a large capacity but slow DCPMM. On the other hand, with the operating system support, both DRAM and DCPMM can be exposed as normal memory and visible to software, tiering memory into fast and slow [15]. We call this a software-managed tiered memory system. In this environment, operating system support is supposed to effectively use both DRAM and DCPMM because the full control is given to software. This paper focuses on system software aspects of tiered memory systems by understanding how the hardware is organized.

2.2 Performance Characteristics

We describe the distinct performance characteristics of a software-managed tiered memory system that runs with the Linux operating system. Figure 1 presents the system organization used in this study. There are two CPU sockets in the system. For each CPU socket, one DRAM node and one DCPMM node are attached. The entire physical address space is comprised of both the DRAM and DCPMM nodes.

In multi-tiered memory systems, the critical factors in performance are not only the *access locality* but also the *access tier* of memory. Figure 2a shows read access latency and bandwidth for each of the four memory nodes measured from MLC [18]. Accessing the local DRAM outperforms the other three memory nodes, which is well established in traditional NUMA architecture. On the other hand, we observe that local DCPMM (L-DCPMM) is slower than that of remote DRAM (R-DRAM) due to the device characteristics. This is in stark contrast with the conventional wisdom that local memory is always faster than remote memory. Note that we also observe a similar pattern in a bandwidth measurement.

Similarly, Figure 2b shows the same type of evaluation over the four CPU (Intel Xeon Gold 6242) sockets with DRAM-only systems. There is no significant difference in latency and bandwidth for access to any remote DRAM nodes. Due to this, placing pages on the remote DRAM nodes in DRAM-only systems is a relatively simple task.

These distinct characteristics motivate us to explore the design space of page management in operating systems. The operating systems need to have the ability to (re)locate memory efficiently and dynamically by understanding the performance characteristics of multi-tiered memory systems. Unlike the DRAM-only systems, not all the remote memory nodes can be considered equal due to the access tier.

2.3 OS Support of Multi-Tiered Memory

The multi-tiered memory hierarchy that is the focus of this paper is distinct from traditional two-tiered memory. Nevertheless, the current Linux still relies on the existing NUMA framework to support multi-tiered memory systems. Such limited OS support makes the page placement sub-optimal in multi-tiered memory architectures. Although we can redefine the NUMA distance table according to the access latency, it does not exploit the full potential of the multi-tiered memory systems. First, Linux classifies memory nodes as either local or remote in a binary way. When promoting or migrating pages, several alternatives among the remote nodes are not considered at all. Second, the Linux does not support demoting (or reclaiming) pages from the upper-tier to the lower-tier memory. In this study, we revisit page placement strategy by considering performance characteristics across access-tier as well as access-locality.

3 Analysis of Page Management to Multi-Tiered Memory Systems

In this section, we investigate existing page management techniques of Linux that have been designed for DRAM-only NUMA systems. Then, we identify the lack of sufficient support for tiered memory systems. Although AutoNUMA [33] can be used for such multi-tiered memory, we observe that it fails to take full advantage of the multi-tiered memory.

3.1 Initial Page Placement

With the introduction of storage class memory (e.g., Optane DCPMM) in main memory, conventional page placement based only on the access locality has a negative impact on performance because the most critical factor in performance is not only the locality but also the memory tier. Meanwhile, current page placement schemes in operating systems have been well established for DRAM-based NUMA architecture, considering the access locality *only* between threads and memory [8, 13, 21]. In Linux, the default page allocation policy tries to use local memory as much as possible to minimize the performance penalty incurred by accessing remote memory. Only if there is no free space in the local memory, the memory allocator looks for free space on a remote memory node known as a *fallback path* [9].

As a result, the default (*local-first*) allocation policy is considered harmful in multi-tiered memory systems. The numbers in Figure 1a present the order used in the default fallback path when a thread runs on CPU-0, considering only the physical distance. If the *local-DRAM* node does not have enough free space, the memory allocator examines the fallback path to determine which memory node the allocation request should be sent. We anticipate that the allocator should ask the *remote DRAM* node to get a free page because this node provides better performance than the local DCPMM node. Surprisingly, however, the fallback path in the state-of-the-art Linux kernel indicates the local DCPMM (lower-tier). It had not taken into account the distinct characteristic that the memory type is more sensitive to performance than the access locality in multi-tiered memory systems.

Problem: If the local DRAM is full, the fallback prioritizes allocating from the local DCPMM (lower-tier), even though the remote DRAM (upper-tier) performs better.

3.2 Dynamic Placement

Although initial page placement plays a crucial role in the given memory space, the decision may not represent optimal performance because it depends on the memory access traffic at runtime. To adjust the placement of pages according to access patterns, the AutoNUMA facility has been included as part of Linux, automatically migrating pages to a mem-

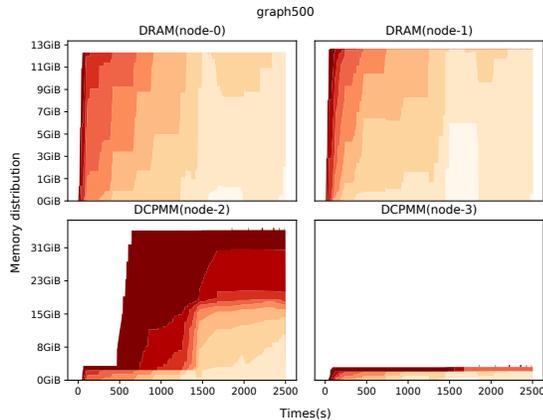


Figure 3: Page distribution and access intensity of `graph500` across memory nodes (Darker colors refer to pages that are accessed relatively more frequently.)

ory node closer to the thread running at runtime [33]. The operating system examines the access locality to find whether the accessed page is placed on the local memory or remote memory. If this is on the remote memory, the page is migrated to the local memory to avoid the remote accesses for subsequent requests. This approach improves the performance of applications running on DRAM-only NUMA systems.

However, we find that the current design does not exploit the advantages of the multi-tiered memory hierarchy. Figure 3 presents memory usage across the memory nodes for `graph500` with the 128GB dataset as time goes by. The detailed experimental setup is explained in Section 5. First, we notice that the upper-tier memory is *ineffectively* used because more frequently accessed pages (dark red) mainly reside in the lower-tier memory (`node-2`). In contrast, less frequently accessed pages are placed on the upper-tier memory¹. The primary reason for this is that the current memory management does not allow page promotion or migration to the upper-tier memory when there is no free space. Although such a design decision is reasonable for DRAM-only systems, we need to reconsider this assumption for multi-tiered memory systems. Figure 4 depicts three cases whereby AutoNUMA encountered page promotion or migration failure due to lack of free space in local DRAM.

Even though the page promotion or migration cannot be made to the best memory node satisfying the access tier as well as the access locality, there are effective alternatives to placement in multi-tiered memory systems. When page promotion fails from remote DCPMM to local DRAM, for example, we have two possible workarounds: placing the page either on the remote DRAM or on the local DCPMM.

Problem: Pages in the lower-tier are not promoted when the upper-tier is fully utilized.

¹Section 4.2.1 explains how we estimate the access frequency.

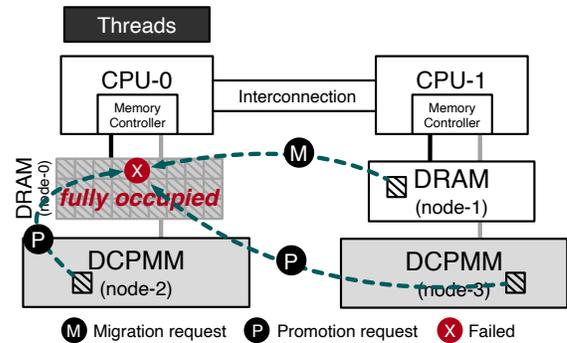


Figure 4: Page promotion and migration failure cases (Promotion: lower-tier \rightarrow upper-tier, Migration: movement between the same tier)

Second, we observe the *skewed* page distribution across the memory nodes in the lower-tier memory. This is because the page movement to CPU-less nodes (DCPMM) is not considered in the current Linux operating systems. Since the traditional OSes were designed under the assumption that memory access performance is highly affected by the access locality between CPU and memory nodes, moving pages to CPU-less nodes does not occur. Only if the destination upper-tier memory has free space, the pages residing in the CPU-less node (lower-tier memory) can be promoted through AutoNUMA. Figure 1a shows arrows all the possible page promotion and migration paths that the stock Linux kernel currently supports. Even though the upper-tier memory is full, so that the operating system cannot place pages on the preferred access tier, we need to be able to preserve the access locality in the lower-tier memory by freely allowing page movement across any CPU-less nodes.

Problem: Pages are never migrated to the CPU-less (lower-tier) nodes due to a NUMA policy that does not apply to multi-tiered memory systems.

3.3 Page Reclamation

The current page reclamation is also designed for DRAM-only systems backed by storage-based swap devices rather than tiered memory systems. Traditionally, `kswapd` reclaims the inactive pages in memory directly to the storage regardless of the memory tier when the memory node is exhausted. It would make sense to reclaim pages in the lower-tier memory to the storage device. However, this is not a desirable solution for the upper-tier memory pages when the lower-tier has enough space. This limitation is intertwined with the two problems explained in Section 3.2.

Problem: Frequently accessed pages from the lower-tier cannot be promoted without demoting less frequently accessed pages from the upper-tier.

Problem / limitation	Our solution	Section
Allocation fallback does not consider the access tier Pages are not promoted when upper-tier is full	Promotion or migration to alternatives	4.1
Pages are never demoted or reclaimed to lower-tier memory	Demotion for the least-accessed pages	4.2
Page classification is too coarse-grained (binary)	Fine-grained access history estimation	4.2.1
Page reclamation to lower-tier & to storage needs to be decoupled	Foreground promotion & background demotion	4.3

Table 1: Problems with current page management implementation, and our solutions

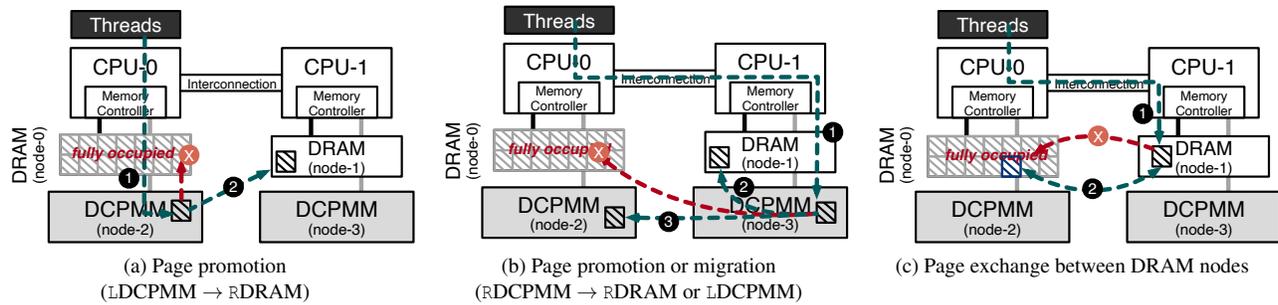


Figure 5: Our conservative design: exploiting multi-tiered memory hierarchy (L: Local, R: Remote)

In Linux operating systems, part of the virtual address space for each process can be mapped, either the `file-backed` or the `anonymous` region. The pages in the `file-backed` region contain the contents of an existing file(s) on memory so that subsequent file I/O operations of the same file can be replaced with memory access. On the other hand, pages belonging to the `anonymous` region do not represent any file contents. This is used for keeping arbitrary data on memory (e.g., `malloc`). For each memory region, the Linux operating system classifies memory pages into `active` or `inactive`. The kernel has inactive lists containing pages that might not be in use while keeping recently accessed pages on the active list. However, the current page classification is too conservative to precisely differentiate which pages are frequently or less frequently accessed from the active and inactive lists.

Problem: Binary page classification (either active or inactive) is too coarse-grained to be used for tiering.

Last, current page reclamation is very carefully performed in the background to hide the cost of accessing the storage devices from the critical path. In tiered memory systems, however, the cost of demoting pages to the lower-tier memory is cheaper than that of swapping out pages to the storage. We need to decouple the demotion to the lower-tier memory from traditional reclaim to the storage.

4 Automatic Multi-Tiered Memory

This section explores the design space of page management to tiered memory systems. The goal of our page management is to extract the full advantage of multi-tiered memory to im-

prove the performance of large-memory applications. To keep our design simple, we base our design on the AutoNUMA facility. Table 1 summarizes the supported mechanisms for each design space. In the following subsections, we explain the design and implementation of each scheme.

4.1 Exploiting Multi-Tiered Memory

As depicted in Figure 4, we take the NUMA fault as a demand signal for page promotion or migration from the DCPMM nodes or the remote DRAM node. Note that the current AutoNUMA deals with the promotion and migration requests only if the upper-tier (local DRAM) memory has free space. Otherwise, the request is discarded, and the faulted page remains in the original memory. When the local DRAM is fully occupied, our proposed design allows the pages to be promoted or migrated into the next best memory node in the multi-tiered memory hierarchy. This approach can exploit the advantage of the multi-tiered memory hierarchy, providing higher performance than the stock Linux kernel.

Figure 5 describes how we react when the local DRAM is full. There are three sources of demand for page migration or promotion to local DRAM in the multi-tier memory system. The multi-tier hierarchy opens up new opportunities to design memory placement. First, (5a) when the faulted page resides in the local DCPMM (1), we promote the page to the remote DRAM as the second best location (2). Since the remote DRAM provides lower latency and higher bandwidth than local DCPMM, we can improve the performance of applications to which memory was initially allocated in the lower-tier memory. Second, (5b) if the faulted page resides in the remote DCPMM (1), we have two alternatives to ex-

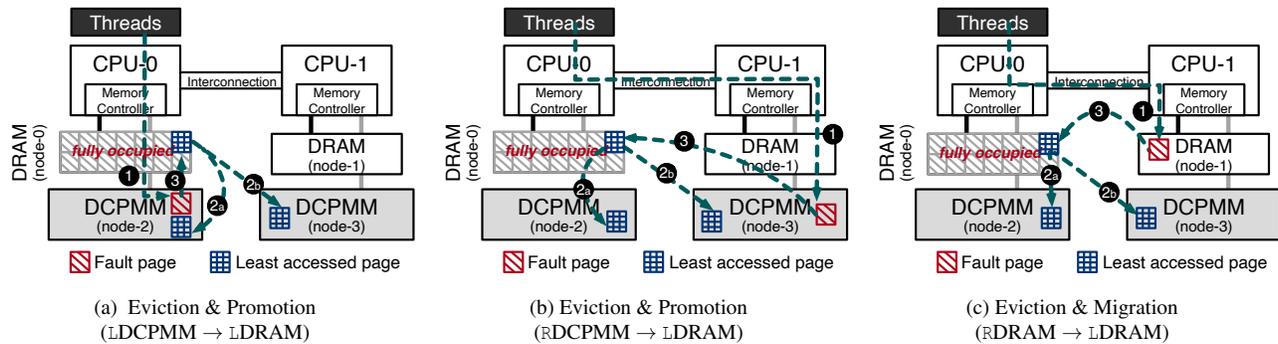


Figure 6: Our progressive design: opportunistic page promotion and migration with page eviction (L: Local, R: Remote)

exploit the advantage of the multi-tier memory hierarchy. We attempt to promote the page to the remote DRAM (2). If the remote DRAM also does not have free space, we try to migrate the page to the local DCPMM (3). Unlike the stock Linux kernel, our modified kernel supports migrating the page to CPU-less nodes (local DCPMM). We call this *AutoTiering Conservative Promotion and Migration (CPM)*. Last, (5c) the faulted page is in the remote DRAM (1). This means that the page is already in the second best location. The existing AutoNUMA is unable to complete the page migration operations between the upper-tier memory nodes. As a result, it leads to sub-optimal performance. In such a case, we consider the page exchange option to satisfy the demand for memory affinity (2). The prior study proposed the page exchange mechanism for tiered memory [35]. We repurpose the mechanism to resolve the migration failures between the same tier memory nodes as well. Internally, for each memory node, we keep track of which pages fail to be migrated. We then leverage this information to determine the migration demand that can be resolved with the exchange operation. We call this *AutoTiering CPM with Exchange (CPMX)*.

Since this design does not require significant changes in the existing Linux operating system, it is easily integrated on top of the AutoNUMA facility. We anticipate that our conservative design can be a practical solution for such software-managed tiered memory systems.

4.2 Opportunistic Promotion and Migration

As we design a conservative approach for finding the best alternative, this is limited to extracting the full performance benefit of software-managed tiered memory. In our conservative design, frequently used pages can reside in the lower-tier (DCPMM) memory while the upper-tier (DRAM) memory holds infrequently accessed data. To relieve such undesirable memory placement, we explore a progressive strategy, opportunistically demoting a page from the upper-tier memory to create free space. This is the main difference between conservative and progressive designs. By demoting a page, the request for page promotion can be successful. For page de-

motion to be effective, we need to have the ability to select a page that is highly unlikely to be reused within a short time. Otherwise, the wrong selection can have a negative impact on performance. We explain how we select a page for the demotion in the following subsection (4.2.1).

Figure 6 depicts how our progressive approach works with page demotion. When the NUMA page fault occurs (1), we find the least accessed page from the upper-tier (local DRAM) memory and compare the access frequency of the least accessed page with the faulted page. If the selected page is relatively less frequently accessed than the faulted page, we demote the selected page to place the faulted page on the higher-tier memory node. Otherwise, we prevent page promotion or migration requests to keep the upper-tier memory with more frequently accessed pages.

To promote a page from either local DCPMM (6a) or remote DCPMM (6b), we demote the least accessed page selected to the lower-tier memory (2a or 2b). The destination of the demoted page depends on where the page was previously accessed to preserve the locality. After that, (3) we can finally promote the page to the local DRAM node. In addition, Figure 6c shows how the page migration request from the remote DRAM is made. We call this *AutoTiering Opportunistic Promotion and Migration (OPM)*.

We further optimize our progressive design by fusing demotion and promotion (or migration) into one exchange operation, called *AutoTiering OPM with Exchange (OPMX)*. When the destination of the demotion is equal to the source of the promotion or migration, we leverage the exchange operation instead of individual promotion and migration. For example (6a), if we need to demote a selected page into the local DCPMM (2a) and promote the page to the local DRAM (3), two individual operations are fused. The exchange operation eliminates unnecessary page allocation and free operations.

In case we cannot find a page to be demoted from the upper-tier memory, we try to promote the page to the next best location - the remote DRAM - as would normally be the case. Since we are conducting page promotion and migration opportunistically, we can reduce excessive page promotion

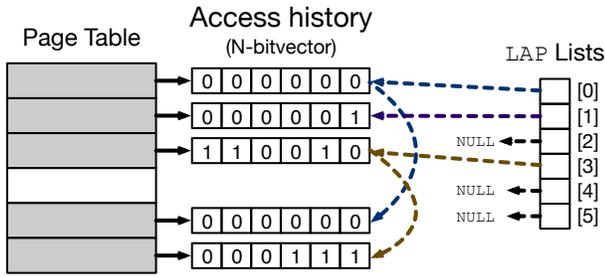


Figure 7: Maintaining least accessed page lists

and demotion that incurs performance overhead associated with page table manipulation and TLB shutdowns.

4.2.1 Predicting the Least Accessed Page

To make the progressive design effective, our goal is to find the least accessed page from the upper-tier memory. As explained in Section 3.3, the Linux operating system separates memory into file-backed and anonymous pages as LRU lists. When page promotion or migration fails due to lack of free space on the upper-tier memory, we investigate the pages from the file-backed region preferentially and move on to the anonymous region if we are unable to find the least accessed page from the file-backed region.

① **File-backed pages:** We examine whether we can make free space by demoting a page belonging to the file-backed region. As file-backed pages are maintained in two LRU lists, active and inactive, we regard the oldest page in the inactive list as the least accessed page. Whenever the file-backed pages are reaccessed (e.g., `sys_read` or `sys_write`), the operating system marks the page as accessed and moves it into the active list. Since the operating system can track the access of file-backed pages, we estimate the least accessed page by looking at the inactive list. If not, we move on to the active list. Note that we preserve the portion of the page cache configured in the kernel parameter (e.g., `vfs_cache_pressure`). If we cannot find a reusable space in the file-backed region, then we look for a page in the anonymous region as a fallback path.

② **Anonymous pages:** On the other hand, we keep the access (fault) information per page to select the least accessed page in the anonymous region judiciously. To minimize the monitoring overhead, we leverage the page scan facility used for AutoNUMA, keeping track of whether the pages are accessed or not during a given time window. Then we build the access history for each page with an N bit-vector. This means that we maintain up to the last N -time access history. We set N to 8. Based on the access history, we classify the pages into N levels (Least Accessed Page lists), where N is the number of bits that are set, as shown in Figure 7. Once page demotion is required, we find one of the pages in the LAP [0] list because those pages have not been accessed the last N times. If the LAP [0] list is empty, then we try to find

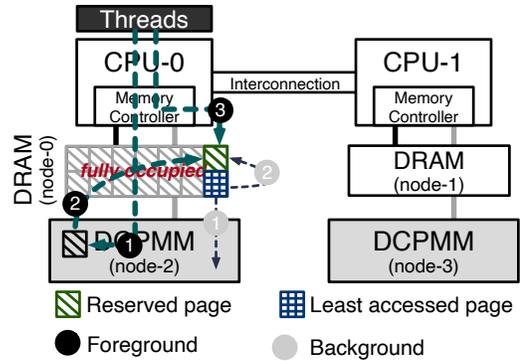


Figure 8: Hiding latency of page demotion with our `kdemoted`

a page from the LAP [1] list, and so on. After that, we can select the least accessed page in the upper-tier memory and conduct page demotion to the lower-tier memory.

4.3 Hiding Latency of Page Demotion

To enforce page promotion, we are supposed to demote a page from the target node. Before completing the page demotion, we cannot proceed with the promotion request due to a lack of space. The fault handling time is the sum of the two operations in the critical path: page demotion and promotion. To remove page demotion from the critical path, we explore a software optimization technique.

We keep a page pool of a few reserved pages. We empirically determine the reserved pages to 16 and 4 for 4KB and 2MB, respectively. The reserved pages allow us to immediately serve the promotion request without requiring the demotion process, even though the upper-tier memory is full. This approach is more cost-effective than the page exchange scheme [35] because it hides the latency of the page demotion in the critical path. Page demotion to the lower-tier memory takes longer than page promotion to the upper-tier memory because the storage-class memory used for lower-tier memory provides better read performance than write performance. To efficiently demote pages in the background, we maintain a new kernel thread called `kdemoted`, demoting the least accessed pages in a batch. Once the number of reserved pages reaches below a certain threshold, we wake the kernel thread to start the demotion process. The threshold is set to 4 through sensitivity studies.

Figure 8 depicts how simple optimization hides the latency of the page demotion. For every promotion request (1), the page can be promoted even when the upper-tier memory is full (2). After completing the promotion, the NUMA fault handler is returned without the demotion process, and future accesses to the page (3) will take place on the upper-tier memory. Meanwhile, `kdemoted` demotes the least accessed pages as needed in a batch (1) to reclaim the memory pool (2). We call this *OPM-BD (Background Demotion)*.

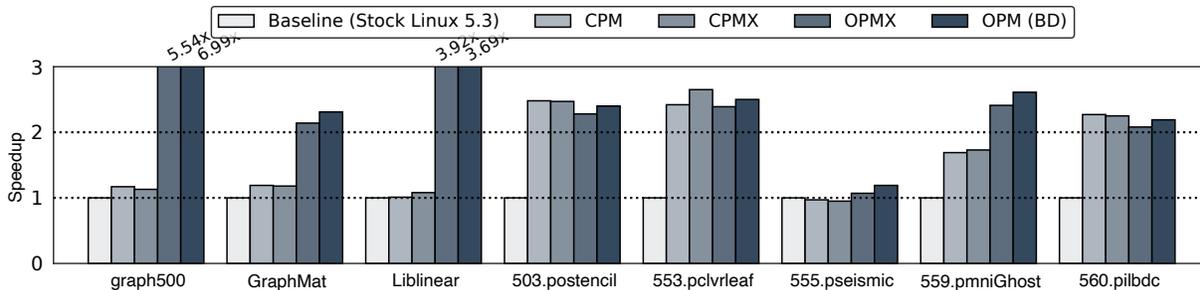


Figure 9: Speedup of our conservative (CPM and CPMX) and progressive (OPMX and OPM (BD)) schemes

5 Evaluation

5.1 Experimental Setup

To evaluate our proposed scheme, we use a NUMA server equipped with two Intel Xeon Gold 5218 processors and compose a multi-tiered memory hierarchy with a 16GB DDR4-2666 DIMM and a 128GB Intel Optane DC Persistent Memory (DCPMM) for each CPU socket. The server system has a total of 32GB DRAM and 256GB DCPMM as the main memory. To minimize measurement variability, we disable HW features, including Hyper-Threading, DVFS, Turbo-Boost, and prefetches. We use the Linux kernel 5.3 and Ubuntu 18.04 server as our baseline and implement our proposed schemes on top of the kernel. Our code is available at <https://github.com/csl-ajou/AutoTiering>. We run benchmarks from graph500, SpecACCEL (OpenMP), GraphMat [32], and Liblinear [22] used in recent large memory systems [2, 35]. We configure them for all the benchmarks to use all 32 cores across two sockets and more than 64GB of memory to sufficiently stress the multi-tiered memory system. Since the page size can affect performance in various ways, we evaluate performance for the large page (2MB) as well as the base page (4KB).

5.2 Experimental Results

Performance with our conservative approach (CPM):

Figure 9 presents the speedup results over the stock Linux kernel (first bar). Note that AutoNUMA is enabled in the default Linux kernel. The second bar in Figure 9 presents the speedup with our conservative promotion and migration (CPM), and the third bar shows performance changes when the conservative exchange is applied (CPMX). For most of the workloads we evaluate, we can see significant performance improvement with our CPM. In 503.postencil, 553.pclvrleaf, and 560.pilbdc, the speedup is over 2x, compared to the baseline. Also, 559.pmniGhost shows 1.6x performance improvement. Our conservative design (CPM) can promote pages from the lower-tier (DCPMM) memory nodes to the remote DRAM node even though the locality is not preserved because the remote DRAM is faster than the local DCPMM. We

can also migrate pages between the two DCPMM nodes to better access locality when accessing the lower-tier memory. As a result, we can utilize the multi-tiered memory systems more efficiently, leading to performance improvement.

For graph500 and GraphMat, we look at that the speedup is around 17% and 19%, respectively. GraphMat and Liblinear read the large dataset from file to its in-memory data structure. The file-backed pages occupy a significant portion of the DRAM nodes, while the crucial data structures reside in the DCPMM nodes. After that, it performs the Pagerank algorithm for analytics. Unfortunately, we could not observe that kswapd is invoked to free the inactive file-backed pages. Thus, our CPM and CPMX do not have the opportunity to utilize the DRAM nodes effectively.

On the other hand, we observe that the performance of 555.pseismic is not improved at all. This shows that the memory placement is well balanced across the upper-tier and lower-tier memory in the baseline. As a result, we could not exploit the promotion and migration opportunities.

Performance with our progressive approach (OPM):

We look at further improvement with our progressive design described in Section 4.2, which finds the least accessed page from the upper-tier (local DRAM) node and demotes that to the lower-tier memory. As shown in Figure 9, OPMX exhibits better performance than CPMX for most of the workloads. In particular, graph500, GraphMat, Liblinear, and 559.pmniGhost show significant speedup over CPMX. The performance of 503.postencil, 553.pclvrleaf, and 560.pilbdc is slightly degraded, by about 7 to 8%. 560.pilbdc fails to amortize the overhead of page promotion and demotion, and 553.pclvrleaf shows that the opportunity for page exchange is reduced.

We analyze the running behavior of graph500 to understand performance improvement. Before performing the core graph algorithm, it generates a lot of intermediate data that occupies the DRAM space. As a result, the baseline and our CPMX spend most of the time accessing the lower-tier memory while running BFS (Breadth-First Search). Interestingly, it exhibits the memory access locality on a small part of the address space, even though the whole address space is huge. With OPMX, we can demote the less frequently accessed data

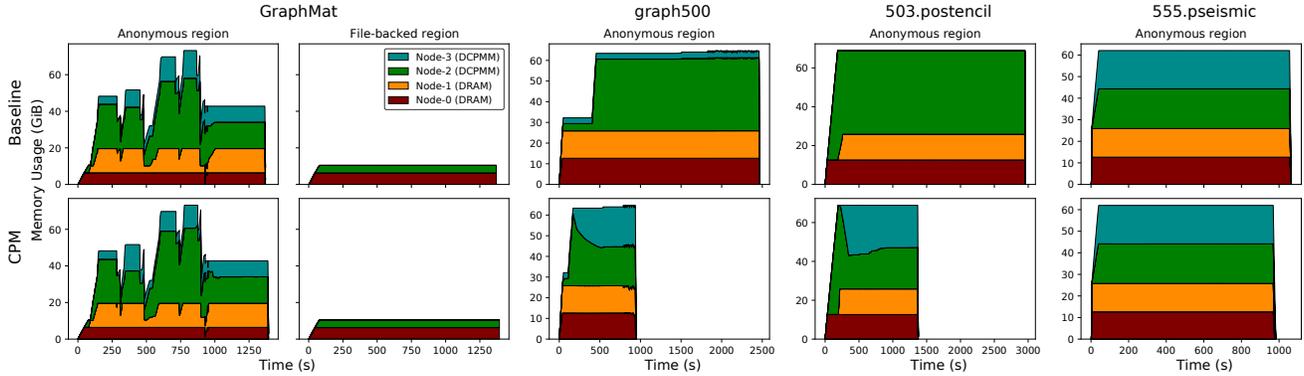


Figure 10: Memory usage across DRAM and DCPMM nodes (Baseline vs. CPM)

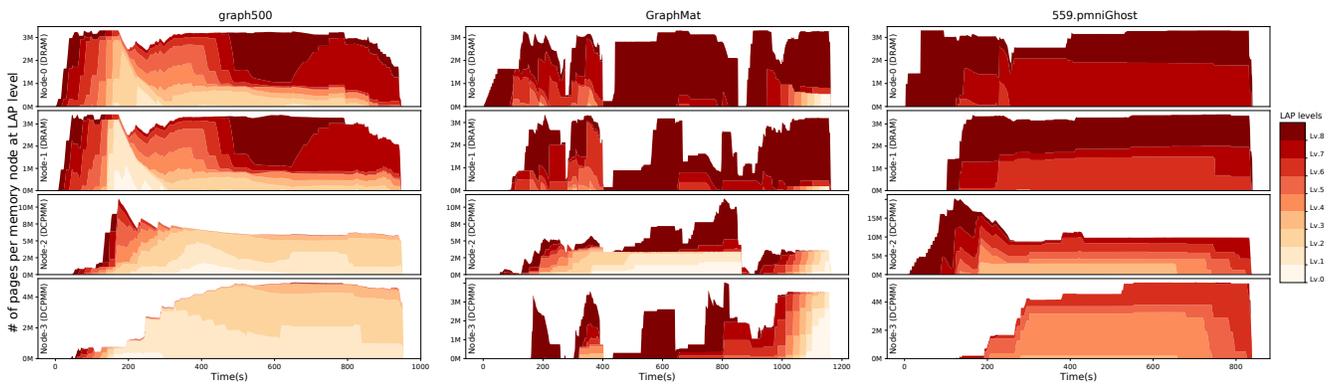


Figure 11: Estimated access frequency of pages corresponding to the LAP levels

to the lower-tier memory and utilize the upper-tier memory space for more frequently accessed data. Therefore, we can increase performance drastically.

For GraphMat, we see significant performance improvement comes from the demotion of file-backed pages. By default, the file-backed page is initially placed on the inactive list. After the page is reaccessed, it moves from the inactive to the active list. By demoting the least accessed page from the file-backed pages, we can improve the utilization of the upper-tier memory.

With OPM(BD), we can further improve the performance of most workloads. The improvement for graph500 and GraphMat is considerable compared to OPMX. 555.pseismic and 559.pmniGhost show marked improvement, and the other SPEC workloads also restore the degraded performance from the exchange version. Meanwhile, Liblinear shows degraded performance slightly.

Distribution of memory usage: We analyze how multi-tiered memory is utilized as time goes by. Figure 10 compares the memory usage for the baseline and our progressive design for selected workloads, which are beneficial from our CPM. For the three SPEC workloads, the lower-tier memory is more well balanced with CPM. This is because we allow pages to be migrated to the CPU-less node. Even though CPM is unable

to satisfy the required access tier, it can preserve the access locality in the lower-tier memory.

For GraphMat, the performance improvement is relatively small compared to the SPEC workloads, and 555.pseismic is not improved. The reason is that the baseline already keeps the memory balance across lower-tier DIMMs. In the next paragraph, we look at the effectiveness of our OPM-based schemes for those two workloads.

Effectiveness of LAP classification: To evaluate the effectiveness of our LAP scheme, we decompose all the pages into each LAP level and build a histogram as time goes by. Figure 11 presents the selected three workloads that show significant improvement with OPMX compared to CPM. For graph500, GraphMat, and 559.pmniGhost, the relatively frequently accessed pages (dark red) corresponding to level 7 or 8 are placed on DRAM nodes, while DCPMM nodes serve the relatively less accessed pages. This result shows that our OPMX can be effective for applications whose working set fits in the upper-tier memory.

The additional space for keeping the access history (8b), two pointers for lists (16B), page frame number (8B), and last faulted CPU number (1B) is required for each page to enable our LAP scheme. Compared to the baseline, we require 32 bytes of metadata per page due to 8 bytes alignment, and it

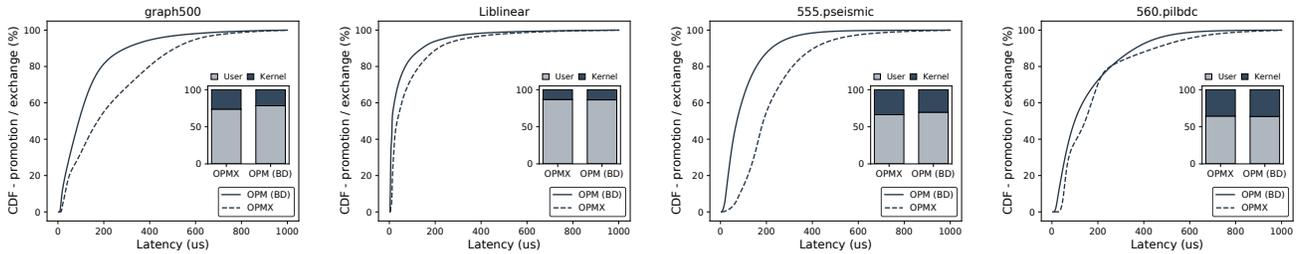


Figure 12: Cumulative distribution function (CDF) of page promotion and migration with OPMX and OPM (BD)

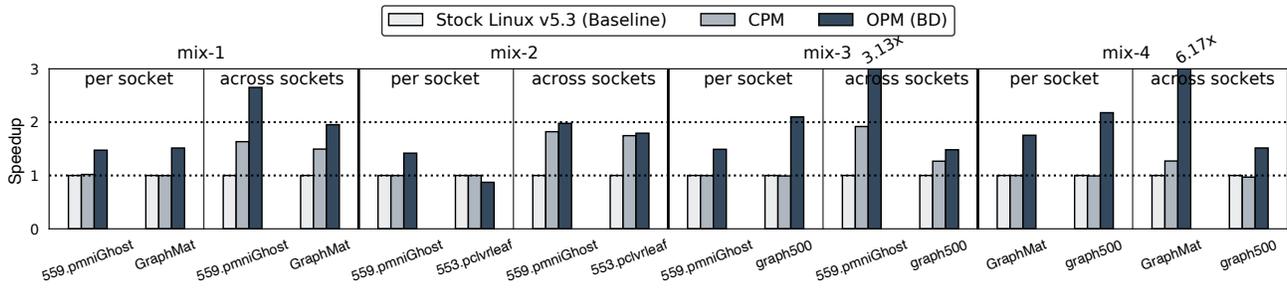


Figure 13: Speedup of multi-programmed execution scenarios

results in an extra 288MB for the system memory of 32GB DRAM with 256GB DCPMM. It slightly decreases the effective DRAM memory space by 0.91%.

Latency hiding with background demotion: We measure promotion and exchange latency distribution in the page fault path with `ftrace` and the kernel and user time. Figure 12 presents the latency CDF serving a page promotion for OPM (BD) and a page exchange for OPMX. The solid line is for the distribution, including the latency hiding scheme (OPM (BD)), and the dashed line is for the OPMX scheme. The fault latency varies for both schemes. We observe that OPM (BD) shows better latency distribution for all the workloads than the exchange version because the demotion is off the critical path. Especially, `graph500` and `555.pseismic` are beneficial to the background demotion.

On the other hand, the end performance of `Liblinear` is not improved with the background demotion shown in Figure 9. Even though the latency is reduced, the kernel execution time is not significantly changed. This is due to the possibility of incurring memory access contentions across the application threads and the background kernel thread. We plan to resolve the undesired background demotion case with rigorous scheduling of `kdemoted` in our future work.

Performance with multi-programmed workloads: We evaluate how well our proposed schemes work for multi-programmed workloads in two scenarios, `per socket` and `across sockets`. Figure 13 shows the speedup with CPM and OPM (BD) when two applications run on the same server. We mimic four multi-programmed scenarios (`mix-1` to 4)

with the combination of the workloads. When isolating the applications based on sockets (`per socket`), CPM does not provide any performance improvements as expected because it lacks the opportunity to exploit the multi-hierarchy of memory. On the other hand, we can observe significant performance improvement with OPM (BD) for all cases except for `553.pclvrleaf` in `mix-2` as more frequently accessed pages can be placed on the upper-tier memory. When allowing the applications to run across sockets, these may not effectively utilize the upper-tier memory nodes because the memory usage across the threads of applications is not evenly distributed. In the `across sockets` setting, we observe that CPM can be useful by exploiting the multi-tiered memory hierarchy. In addition, OPM (BD) can further improve performance, compared to CPM, for all the cases.

Working-set sensitivity: Figure 14 presents the performance by varying the working-set size from 32GB to 160GB for selected workloads. We observe that the performance of `graph500` and `559.pmnigHost` is significantly improved by both CPM and OPM (BD). For `503.postencil` and `553.pclvrleaf`, however, OPM (BD) and CPM show similar performance improvements as the working-set size increases. As explained in the LAP classification paragraph, most of the pages are evenly accessed in those benchmarks less beneficial to OPM (BD). Note that the speedup with our approach is significant and still effective compared to stock Linux (Baseline). Since the other workloads, such as `GraphMat` and `Liblinear`, have fixed problem input sizes, we could not evaluate the working-set sensitivity for the workloads.

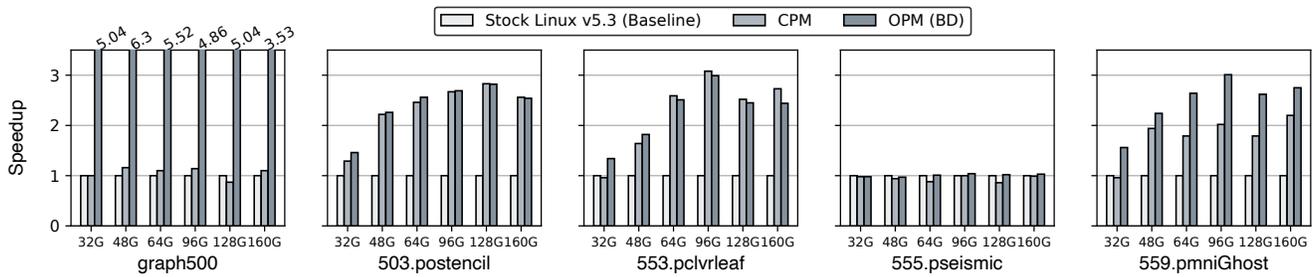


Figure 14: Performance speedup by varying the working-set size from 32GB to 160GB

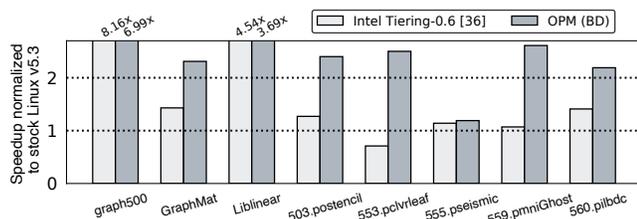


Figure 15: Performance comparison: OPM(BD) and Tiering v0.6 [36]

Performance comparison with prior studies: Figure 15 presents the performance comparison results with a recent proposal from Intel called Tiering v0.6 [36]. Note that Tiering v0.6 is based on Linux kernel version 5.9 but not merged into the mainline. We show that our OPM(BD) outperforms the performance of Tiering v0.6 for most of the workloads.

For the anonymous memory region, Tiering v0.6 supports the page promotion to the upper-tier memory by extending the AutoNUMA framework. Through the monitoring facility, they investigate whether the page is accessed during the last two consecutive scans or not. If so, they consider the page is hot and promote it. On the other hand, our OPM(BD) maintains access history for the last N (8) times. The decision is more accurate than looking at the previous two accesses.

Besides, Tiering v0.6 inherits the same limitation from the traditional AutoNUMA. Once the local DRAM becomes full, it is allowed to promote the page to neither the remote DRAM nor the local DCPMM. Instead, `kswapd` is triggered to reclaim pages to lower-tier memory. In contrast, our LAP scheme makes the upper-tier memory better utilized by opportunistically performing promotion and demotion.

For `graph500`, we observe that Tiering v0.6 performs better than OPM(BD). Our scheme outperforms the time for building graphs before execution, but while traversing the graph, `graph500` with OPM(BD) accesses more pages in the lower-tier memory compared to Tiering v0.6. In `Liblinear`, it shows that Tiering v0.6 more aggressively demotes the file-backed pages, leading to better utilization of the upper-tier memory.

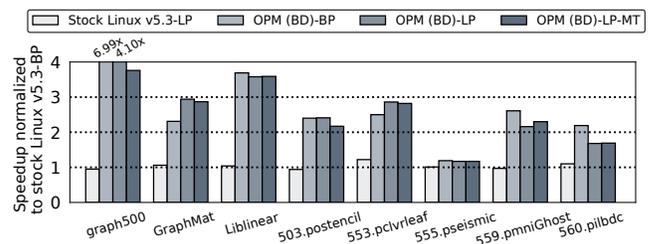


Figure 16: Speedup with large page (LP) over base page (BP)

Performance with large page: To minimize the overhead of TLB misses for large memory applications, modern computer systems provide large page options. Figure 16 shows the speedup results when we leverage the large page (2MB) instead of the base page size (4KB). For `GraphMat` and `553.pclvrleaf`, the performance improvement with large pages can be observed in the OPM(BD) scheme. On the other hand, most of the workloads do not present a significant difference. `559.pmniGhost` and `560.pilbdc` exhibit degraded performance. When analyzing the LAP histogram for the SPEC workloads with the large page, our scheme can quickly separate pages into the LAP levels compared to the base page. However, there is performance degradation because the overhead of page demotion increases when moving the large pages. To reduce the overhead, we take advantage of the multi-threaded (MT) version of copying pages [35]. `559.pmniGhost` only shows improved performance, but the other workloads become even worse. As mentioned in the earlier paragraph (*Latency hiding*), we will further investigate how our scheme can be extended to mitigate the performance overhead in our future work.

6 Related Work

There have been significant efforts throughout hardware and software to use tiered (or heterogeneous) memory systems effectively. We compare our scheme with prior approaches. First, most previous studies focused on designing page migration mechanisms and policies on two-tiered memory systems [3, 10, 11, 19, 20, 24, 26, 29, 35]. Unlike the prior work, this study extends the problem space to the tiered memory

augmented on traditional NUMA architecture, called multi-tiered memory systems. As there are several alternatives to page placement in multi-tiered memory systems, this paper exploits such opportunities when placing, promoting, and demoting pages. Second, *AutoTiering* does not differentiate pages into hot and cold with a predefined threshold used in prior work [3, 19, 20]. In this study, promotion and demotion decisions are made on the relative access frequency and recency across the memory tier. To estimate page access activities, we rely on the AutoNUMA facility. Last, we use a real-world infrastructure to evaluate our proposed ideas based on Intel's Optane Persistent Memory (DCPMM), which has attracted recent attention. Prior work emulated or simulated two-tiered memory systems based on DRAM [3, 11, 19, 20, 35]. Although real-world storage-class memory (SCM) shows asymmetry in read and write performance, it was not correctly modeled in emulating tiered memory with DRAM.

Below, we describe previous software efforts in the OS community. To understand the heterogeneity of memory systems in Linux, the ACPI 6.2 specification introduced heterogeneous memory attributes tables (HMAT) to provide users with performance information for various memory types [39]. Since the Linux kernel 5.0-rc1, the persistent memory (here, Intel's DCPMM) can be used as volatile main memory, although it is slower than DRAM [15]. It can provide abundant main memory space, but the policy and mechanism supporting tiered memory hierarchy are in infancy. Recently, a new memory allocator for hardware-managed DRAM cache known as shuffle page allocator introduced in the Linux kernel [34]. However, it is not enough to extract the full performance of tiered memory because it does not consider the distance between memory nodes and NUMA typologies. Also, there have been efforts to efficiently support page migration between fast and slow memory, but these still rely on active and inactive list management [5, 16, 30] and have not been merged into the mainline of the Linux kernel until now. In the Windows operating system, they measure the cost of various page operations when the system is initialized through `MiComputeNumaCost` and build a table like the NUMA distance of Linux, but this reflects the access costs [37]. Due to limited information for Windows OS, we could not find how they work for multi-tiered memory.

Researchers in the architecture community introduced hardware techniques to effectively utilize two-tiered memory systems while minimizing the performance overhead in estimating access frequency [7, 28, 29, 31]. Choe et al. suggested memory allocation schemes for the hardware-assisted multi-tiered memory systems where the DRAM nodes are invisible to software [6]. Except for that, none of the work considered the case of multi-tiered memory systems. The overheads such as tracking and migrating pages can be reduced significantly by architectural support, but the flexibility of making decisions for promotion and demotion considering placement alternatives is limited.

7 Conclusion

This work explored a set of new page management schemes called *AutoTiering*, which benefit from multi-tiered memory systems. We found that the Linux operating system focuses on the access locality posed by NUMA, rather than the memory tier. However, in multi-tiered memory systems, the cost of accessing memory is not proportional to solely the locality. We comprehensively addressed the diverse aspects of utilizing the multi-tiered memory systems by considering two factors: the access tier and locality. We built a proof of concept with a real-world tiered memory system. Our evaluation showed significant performance improvements in various benchmarks than the stock Linux kernel version 5.3 and the previous approach from Intel's Tiering v0.6.

Future tiered-memory systems are expected to be more diverse and heterogeneous. To make our approach more general, we can maintain a table describing possible alternatives to placement. While initializing memory in the operating system, we can measure actual performance across memory nodes. With such a new table, *AutoTiering* can adjust where pages need to be promoted, demoted, or migrated adaptively, as explained, without a static decision.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2019R1C1C1005166).

References

- [1] Samsung unveils industry-first memory module incorporating new cxl interconnect standard, 2021. <https://bit.ly/3uBo27J>.
- [2] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently self-replicating page-tables for large-memory machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.
- [3] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
- [4] Microsoft Azure. Introducing new product innovations for sap hana, expanded ai collaboration with sap and more, 2019. <https://azure.microsoft.com/ko-kr/blog/introducing-new->

[product-innovations-for-sap-hana-expanded-ai-collaboration-with-sap-and-more/](#).

- [5] Keith Busch. Page demotion for memory reclaim, Mar 2019. <https://lwn.net/Articles/783672/>.
- [6] Wonkyo Choe, Jonghyeon Kim, and Jeongseob Ahn. A study of memory placement on hardware-assisted tiered memory systems. *IEEE Computer Architecture Letters (CAL)*, 19(2), 2020.
- [7] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.
- [8] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.
- [9] Linux Kernel Documentation. What is numa?, June 2019. Available at <https://www.kernel.org/doc/Documentation/vm/numa>.
- [10] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurusurthi, and Ada Gavrilovska. Kleio: A hybrid memory page scheduler with machine intelligence. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2019.
- [11] 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 Eleventh European Conference on Computer Systems (EuroSys)*, 2016.
- [12] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on numa systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2014.
- [13] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. Challenges of memory management on modern numa systems. *Communication of ACM*, November 2015.
- [14] Google. Available first on google cloud: Intel optane dc persistent memory, 2019. <https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory>.
- [15] Dave Hansen. Allow persistent memory to be used like normal ram, Jan. 2019. <https://lwn.net/Articles/776921/>.
- [16] Dave Hansen. [RFC] Memory Tiering, 2019. Available at <https://lore.kernel.org/linux-mm/c3d6de4d-f7c3-b505-2e64-8ee5f70b2118@intel.com/>.
- [17] Intel. Baidu feed stream services restructures its in-memory database with intel optane technology, 2019. <https://newsroom.intel.com/wp-content/uploads/sites/11/2019/08/baidu-feed-case-study.pdf>.
- [18] Intel. Intel memory latency checker v3.9, 2020. <https://software.intel.com/en-us/articles/intelr-memory-latency-checker>.
- [19] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.
- [20] 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 Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.
- [21] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and memory placement on numa systems: Asymmetry matters. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2015.
- [22] Chih-Jen Lin. Liblinear – a library for large linear classification. <https://www.csie.ntu.edu.tw/~cjlin/liblinear/>.
- [23] Lily Looi and Jianping Jane Xu. Intel optane data center persistent memory. In *HotChips : A Symposium on High-Performance Chips (HotChips)*, 2019.
- [24] Jeffrey C. Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. Operating system support for nvm+dram hybrid main memory. In *Proceedings of*

the 12th Conference on Hot Topics in Operating Systems (HotOS), 2009.

- [25] Oracle. Oracle and intel collaborate on optane dc persistent memory performance breakthroughs in next generation oracle exadata x8m, 2019. <https://www.oracle.com/corporate/pressrelease/oow19-oracle-intel-partner-optane-exadata-091619.html>.
- [26] Mark Oskin and Gabriel H. Loh. A software-managed approach to die-stacked dram. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT)*, 2015.
- [27] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramcloud. *Communication of ACM*, 54(7), July 2011.
- [28] Moinuddin K. Qureshi and Gabe H. Loh. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.
- [29] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2011.
- [30] Yang Shi. Migrate mode for node reclaim with heterogeneous memory hierarchy, Jun 2019. Available at <https://lwn.net/Articles/791174/>.
- [31] Jaewoong Sim, Alaa R. Alameldeen, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim. Transparent hardware management of stacked dram as part of memory. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.
- [32] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, July 2015.
- [33] Rik van Riel and Vinod Chegu. Automatic numa balancing, 2014. Available at https://www.redhat.com/files/summit/2014/summit2014_riel_chegu_w_0340_automatic_numa_balancing.pdf.
- [34] Dan Williams. mm: Randomize free memory, January 2019. <https://lwn.net/Articles/776228/>.
- [35] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 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.
- [36] Huang Ying. tiering-0.6, 2020. Available at <https://git.kernel.org/pub/scm/linux/kernel/git/vishal/tiering.git/commit/?h=tiering-0.6>.
- [37] Pavel Yosifovich, Mark Russinovich, David Solomon, and Alex Ionescu. *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more (Developer Reference)*. Microsoft Press, 2017.
- [38] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.
- [39] Ross Zwisler. Add support for heterogeneous memory attribute table, June 2017. <https://lwn.net/Articles/724562/>.

A Fast and Flexible Hardware-based Virtualization Mechanism for Computational Storage Devices

Dongup Kwon, Dongryeong Kim, Junehyuk Boo, Wonsik Lee, and Jangwoo Kim*
*Department of Electrical and Computer Engineering
Seoul National University*

Abstract

A *computational storage* device incorporating a computation unit inside or near its storage unit is a highly promising technology to maximize a storage server’s performance. However, to apply such computational storage devices and take their full potential in virtualized environments, server architects must resolve a fundamental challenge: *cost-effective virtualization*. This critical challenge can be directly addressed by the following questions: (1) how to virtualize two different hardware units (i.e., computation and storage) and (2) how to integrate them to construct virtual computational storage devices, and (3) how to provide them to users. However, the existing methods for computational storage virtualization severely suffer from their low performance and high costs due to the lack of hardware-assisted virtualization support.

In this work, we propose *FCSV-Engine*, an FPGA card designed to maximize the performance and cost-effectiveness of computational storage virtualization. FCSV-Engine introduces three key ideas to achieve the design goals. First, it achieves high virtualization performance by applying *hardware-assisted virtualization* to both computation and storage units. Second, it further improves the performance by applying *hardware-assisted resource orchestration* for the virtualized units. Third, it achieves high cost-effectiveness by *dynamically constructing and scheduling* virtual computational storage devices. To the best of our knowledge, this is the first work to implement a hardware-assisted virtualization mechanism for modern computational storage devices.

1 Introduction

A modern *computational storage* device incorporating a computation unit inside or near its storage unit is becoming a highly promising solution for high-performance storage servers as it can minimize the data movement overhead with *near-storage processing* [10, 15, 17, 21, 27, 33, 35, 39]. A server equipped with a computational storage device can offload

*Corresponding author.

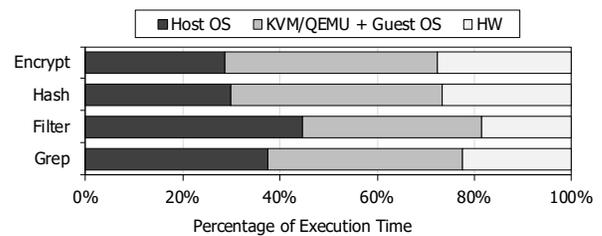


Figure 1: Percentage of execution time spent on a paravirtualized computational storage device.

data-intensive routines to the computation unit and enable it to access the data stored in the storage unit without software intervention. For example, current off-the-shelf computational storage devices incorporate high-end field-programmable gate arrays (FPGAs) for computations and adopt an NVM Express (NVMe) storage protocol to allow the FPGAs to directly access the solid-state drives (SSDs) [6, 24, 33, 35].

In addition to the advances in the hardware devices, recent studies have proposed flexible FPGA overlay architectures and POSIX-like APIs to improve the usability of modern computational storage devices [33, 35]. Their stream-based overlay architectures contain user-specified operators (e.g., stream processing units) and activate them selectively through an in-FPGA crossbar switch and control logic. At the same time, their custom software stacks provide abstraction layers to hide the complexity of the underlying FPGA implementations. For example, a recent computational storage implementation takes advantage of Linux file and pipe abstractions to allow users to easily orchestrate their near-storage processing [35].

However, the existing virtualization mechanisms for computational storage severely suffer from their low performance and high costs. First, the software-based virtualization mechanisms (e.g., paravirtualization) cannot take full advantage of near-storage processing due to their (1) heavy hypervisor and host OS stacks to emulate virtual computational storage devices and (2) indirect resource orchestration mechanisms via guest and host OSes. To profile the software overhead of paravirtualized computational storage devices, we measured the

end-to-end execution time of near-storage processing benchmarks on a virtual machine (VM). The near-storage processing benchmarks read 4-KB pages from an NVMe SSD and perform four different stream operations (*encryption, hash, filter, grep*) on an FPGA. Figure 1 shows the percentage of execution time spent on the paravirtualized near-storage processing benchmarks. The hypervisor and guest/host OSes account for a significant portion of their overall execution time (72%–81%) due to the software-centric device emulation and resource orchestration.

Second, the existing virtualization mechanisms which statically allocate both computation and storage units for each VM will incur high hardware costs due to the inefficient use of the shared device resources. (1) The existing SSD-FPGA coupled architectures suffer from their limited scalability when multiple VMs share a single computational storage device. For example, concurrently running many VMs and their near-storage processing workloads on a single computational storage device result in the performance bottleneck at the single shared SSD. (2) Moreover, their static resource allocation methods cannot handle the dynamic behavior of VM workloads efficiently, which incurs the extra costs for the additional hardware resources to meet quality-of-service (QoS) requirements.

In this paper, we propose FlexCSV, a new hardware virtualization mechanism to maximize the performance and cost-effectiveness of computational storage virtualization. FlexCSV combines the following key ideas. First, FlexCSV implements *hardware-assisted virtualization and resource orchestration*. Through a standard single-root I/O virtualization (SR-IOV) layer at the hardware level, FlexCSV provides a fast and host-bypassing virtualization stack and allows multiple VMs to exploit near-storage processing capabilities. In addition, FlexCSV manages user-requested stream operations at the hardware level to mitigate the software burden to orchestrate near-storage processing.

Second, FlexCSV achieves high cost-effectiveness by *dynamically constructing and scheduling both computation and storage units*. To improve its scalability, FlexCSV adopts an SSD-FPGA decoupled architecture and allows the FPGA accelerator card to construct many virtual computational storage devices with multiple PCI Express (PCIe) attached SSDs. Moreover, its dynamic resource allocation from a shared hardware operator pool and partial reconfiguration support can capture the dynamic behavior of VM workloads and reduce QoS violations significantly at minimum hardware costs.

For evaluation, we implemented our FlexCSV prototype on a Xilinx FPGA accelerator card [7], NVMe SSDs [3], and an existing KVM/QEMU virtualization stack [4, 5]. We implemented a hardware-assisted virtualization stack for computational storage on the same FPGA card and connected its hardware modules through advanced extensible interface (AXI) interconnects. An in-FPGA AXI crossbar switch orchestrates data movements between the processing units and

the FPGA’s on-board DRAM. In this work, we implemented eight hardware near-storage processing operators and allowed them to be shared by four VMs.

Our experimental results show that our FlexCSV prototype obtains 2.0x–2.8x higher near-storage processing performance in virtualized environments than the existing software-centric virtualization mechanisms. Moreover, FlexCSV’s SSD-FPGA decoupled architecture can connect four PCIe-attached SSDs and provide 3x more scalable performance over the coupled computational storage architectures. Through its dynamic resource allocation for both computation and storage units, FlexCSV can reduce QoS violations significantly at minimum hardware costs when a computation storage device is oversubscribed by many VMs.

In summary, we make the following contributions:

- **Novel virtualization stack for computational storage:** We propose a fast and flexible hardware-assisted virtualization mechanism for modern computational storage devices.
- **High performance:** FlexCSV achieves high virtualization performance by bypassing software stacks and leveraging near-storage processing.
- **High cost-effectiveness:** FlexCSV achieves high hardware cost-effectiveness by dynamically constructing and scheduling both computation and storage units at minimum costs.
- **Prototyping:** We implement and evaluate our FlexCSV prototype with off-the-shelf devices and open-source software virtualization stacks.

2 Background

2.1 SSD-FPGA Computational Storage

2.1.1 Hardware Architecture

SSD-FPGA hardware platform. A modern computational storage device incorporates its computation and storage units together and couples them through an on-board interconnect [24, 33, 35]. Figure 2 (bottom) shows the typical hardware platform of modern SSD-FPGA computational storage devices. It utilizes an FPGA for computations and allows it to directly access an attached NVMe SSD. For near-storage data processing, it also supports peer-to-peer (P2P) data communications between the computation and storage units through its internal switches (e.g., on-board PCIe switch). For example, by exposing an FPGA’s DRAM space to an SSD and implementing a routing policy in an internal crossbar switch, a computational storage device can enable the computation and storage units to exchange data directly without software arbitration [35].

FPGA overlay architecture. The FPGA overlay architectures implemented on computational storage devices offer programmable near-storage processing [33, 35]. They consist

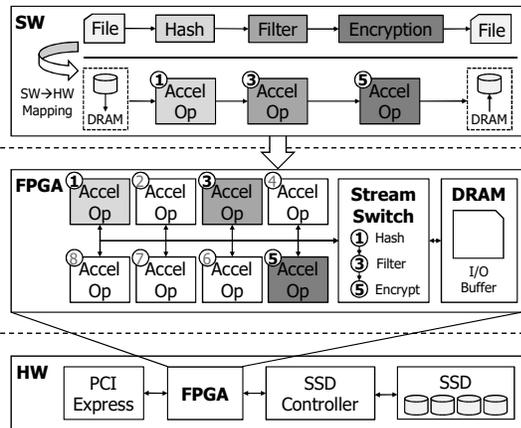


Figure 2: Full software and hardware stacks for a modern FPGA-based computational storage device.

of reconfigurable operators, crossbar stream switches, and on-board DRAM. The *operators* are reconfigurable stream processing units, and each of them can be selectively activated to configure full stream operations. The on-board crossbar switches orchestrate data movements between the operators and the FPGA’s DRAM. Figure 2 (middle) shows an example overlay architecture for programmable near-storage processing. This overlay architecture contains eight operators and three of them are activated to serve the user-specified stream workloads (hash, filter, encryption). Then, a routing policy is installed in the stream switch to properly route an incoming data stream to the target operators (hash→filter→encryption) or DRAM buffers.

The FPGA overlay architectures also implement storage interfaces to allow their operators to directly access storage units [24,33,35]. Modern computational storage devices adopt an NVMe standard storage protocol to offer fast and parallel storage access [24]. NVMe has the following key advantages. First, NVMe supports multiple I/O queues to fully utilize the high-bandwidth storage units. NVMe can run multiple storage operations concurrently by assigning separate NVMe submission queue (SQ)/completion queue (CQ) pairs to different processing cores. Second, NVMe enables fast storage operations by minimizing the number of memory-mapped I/O (MMIO) operations. For example, NVMe devices expose a set of SQ/CQ doorbell registers and require only a single MMIO write operation (i.e., doorbell register write) to submit storage operations or to notify their completions.

2.1.2 Software Support

Abstraction layer. To improve the usability of near-storage processing in modern computational storage devices, recent studies provide custom software stacks utilizing Linux file and pipe abstractions and allow users to easily orchestrate near-storage processing [33,35]. Their software stacks expose the operators implemented on their FPGA overlay architec-

```

1  typedef struct {
2      ap_uint<64> data;
3      ap_uint<4> dest;
4      ap_uint<1> last;
5  } stream_data;
6  typedef hls::stream<stream_data> stream_t;
7
8  void accel_unit(
9      stream_t &in, stream_t &out, ap_uint<4> dest){
10     stream_data input, output;
11     do {
12         input = in.read();
13         output.data = process(input, data);
14         output.dest = dest;
15         out.write(output);
16     } while (!input.last);
17 }

```

Listing 1: Example HLS code for an operator implemented on an FPGA overlay architecture.

tures as executable files to an OS. Then, a user program can initiate near-storage processing through POSIX-like APIs or a pipe command from data to executable (i.e., operator) files. Figure 2 (top) shows an example user program and its software-to-hardware mapping process. In this example, the application performs a hash→filter→encryption stream operation on the input file stored in the computational storage device. In this way, the software support can hide the complexity of the underlying FPGA implementations and coordinate user-specified near-storage data processing.

High-level synthesis support for operators. The software support also allows users to customize operators using an FPGA’s reconfigurability and high-level synthesis (HLS) tools [35]. Listing 1 shows an HLS code snippet to implement an example `accel_unit` operator. First, users can define a stream data structure. In this example, `dest` and `last` signals are delivered along with a data stream. The `dest` field indicates its next destination operator and the `last` field indicates a last word in the data stream. Second, users can define the input and output ports of an operator. In this example, the `accel_unit` module has the `in/out` data stream ports and the `dest` configuration register to determine its next destination operator. This example operator reads a word through the `in` stream port and forwards it through the `out` stream port after manipulating the `dest` field of the output stream.

2.2 I/O Virtualization

2.2.1 Software-based Virtualization

The existing software-based I/O virtualization presents virtual device instances and enables device sharing across multiple VMs. *Full virtualization*, which is one of the software-based virtualization mechanisms, utilizes a trap-and-emulate approach to provide virtual device instances to VMs without changing the guest OSes. However, this virtualization mechanism significantly suffers from excessive VM exits when

guest OSes access their virtual device resources.

In contrast to full virtualization, *paravirtualization* enables efficient virtual device emulation by creating VM-friendly virtual device interfaces between guest OSes and hypervisors (e.g., virtio [34]). This paravirtualization mechanism incurs a fewer number of VM exits by minimizing MMIO operations for device access, but it still relies on software traps to a hypervisor and CPU mode switches. Moreover, guest OSes should be aware that they are being virtualized and modified to interact with a hypervisor in this efficient manner.

To virtualize modern fast and high-bandwidth devices, recent *sidecore approaches* dedicate multiple CPU cores for device emulation [29, 38]. As dedicated sidecores in the host software keep polling guest I/O operations via shared memory regions, VMs do not have to incur VM exits to submit device operations. In this way, the sidecore approaches minimize the performance overhead incurred by VM exits and CPU context switches [23]. However, the sidecore approaches demand a large amount of computing resources of a host server machine to execute their polling-based device emulation [22, 25, 29].

2.2.2 Hardware-assisted Virtualization

To overcome the performance overhead and host inefficiency of the software-based virtualization mechanisms, hardware-assisted virtualization mechanisms allow guest OSes to access target PCIe devices directly without any software arbitration. To enable the direct assignment of PCIe devices (i.e., *passthrough* virtualization), an I/O memory management unit (IOMMU) and its direct memory access (DMA) and interrupt remapping mechanisms are introduced. The DMA remapping mechanism allows DMA operations from virtual devices to be accomplished with guest physical addresses. Similarly, the interrupt remapping mechanism translates interrupt vectors caused by the virtualized devices into VM contexts. However, this approach requires a physical device to be exclusively assigned to a single VM and does not support device sharing across multiple VMs.

To address such shortcomings, PCIe SR-IOV allows a physical device to be shared by many VMs at the hardware level. An SR-IOV capable device presents multiple physical functions (PFs) and virtual functions (VFs) (i.e., virtual device instances) at the device interface. Since VFs have separate PCIe configuration registers, including base address registers (BARs), SR-IOV can enforce resource isolation while serving multiple VMs. Moreover, an SR-IOV capable device implements how to multiplex itself at its internal bridge module and thus does not rely on any host software to multiplex its virtual device instances.

3 Motivation

The increasing density and performance of modern computation and storage devices create new opportunities for

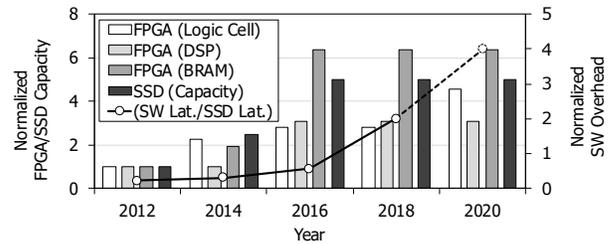


Figure 3: Increasing resource density and performance of FPGA and SSD products over time [3, 8]. The data are normalized to the values at 2012. The latency of the SW stack is set to 20 μs [30, 33], and the current SSD access time is assumed to be 5 μs based on recent studies [33, 41].

virtualization. Figure 3 shows the increase in SSD capacity and FPGA resources over the recent eight years. The available commodity SSD and FPGA resources have increased 3x–6x over the years, which creates a high potential to serve many VMs on a single computational storage device [19, 22, 25, 29, 40]. Furthermore, the significant improvements in storage device access time (65 μs \rightarrow 5 μs [33]) have moved the performance bottlenecks to the software stacks.

However, the existing software-based SSD and FPGA virtualization mechanisms have the following limitations when providing near-storage processing between the virtualized computation and storage units. First, the software-based virtualization mechanisms cannot take full advantage of near-storage processing due to their indirect device-control and data paths to emulate virtual computational storage devices. Second, their static resource allocation for each VM incurs high hardware costs due to the inefficient use of the shared device resources.

3.1 Indirect Device-Control and Data Paths

Employing software-centric virtualization to SSD and FPGA devices separately suffers from indirect device-control and data paths between the virtualized hardware units and fails to take full advantage of near-storage processing. To measure the software overhead of paravirtualized SSD and FPGA operations, we implemented a virtio-based virtualization stack on KVM/QEMU and profiled the end-to-end execution time of SSD-FPGA near-storage processing. Figure 4 shows two software-centric implementations for computational storage virtualization. In the full software implementation, SSD and FPGA operations involve VM exits and traps to a hypervisor. In the optimized software implementation, accessing an SSD from a VM can bypass the hypervisor and host OS stacks through to an IOMMU and SR-IOV support. However, utilizing an FPGA still relies on the software-centric virtualization mechanisms and suffers from the software-side performance overhead. Moreover, since a guest OS cannot obtain host physical addresses of the FPGA’s BARs, its input and output data must be transferred via the guest and host OS stacks.

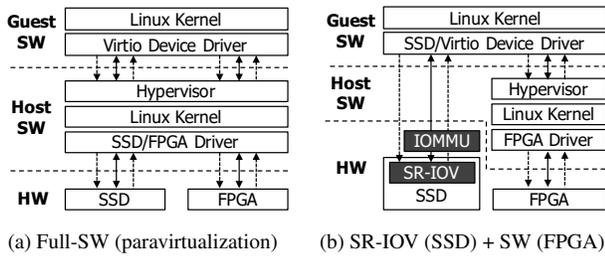


Figure 4: Software-centric virtualization mechanisms for computational storage.

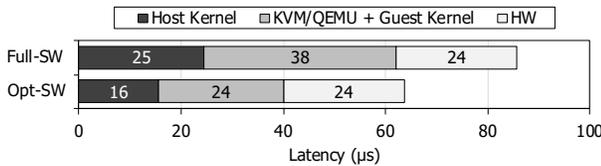


Figure 5: Execution time spent on a paravirtualized computational storage device with SW-centric virtualization implementations.

Figure 5 shows the execution time spent in paravirtualized SSD-FPGA near-storage processing. The benchmark reads a 4-KB page from the NVMe SSD and executes Blowfish encryption [28] on the FPGA. To emulate a coupled computational storage device, we installed an Intel Optane SSD [3] and Xilinx Alveo U250 FPGA [7] and connected them through PCIe Gen-3 lanes. The SSD read latency is $10 \mu\text{s}$, and the encryption operation takes $14 \mu\text{s}$ to process the 4-KB data. In this environment, the hypervisor and guest/host OSes account for a significant portion of overall execution time due to the software-centric device emulation and resource orchestration. The optimized software virtualization implementation mitigates host OS overhead because the SSD access bypasses the host software. Also, as there is no need for KVM/QEMU to emulate the SSD, the hypervisor and guest kernel overhead also decrease. However, the overhead for the virtual FPGA emulation and the data movements between the virtualized units still remains.

3.2 SSD-FPGA Coupled Architecture

An SSD-FPGA coupled computational storage architecture severely suffers from its limited scalability due to its board-level SSD-FPGA integration [6, 17, 33, 35]. Such tight device integration makes it challenging to merge diverse SSD-FPGA resource combinations into a single device while supporting direct device-control and data paths among all the consolidated devices. Moreover, their architectural limitations become increasingly apparent as the gaps between SSD and FPGA resource capacity and performance increase.

For example, concurrently executing many VMs and their near-storage processing workloads on a computational stor-

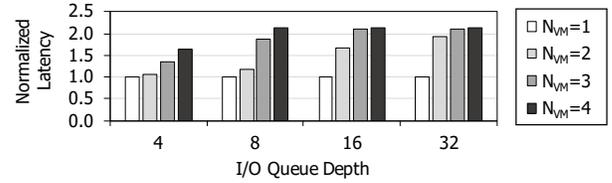


Figure 6: Performance comparison of near-storage processing with the increasing number of SSD-contending VMs and queue depth. The results are normalized to the single-VM latency. N_{VM} indicates the number of concurrently running VMs.

age device can lead to the performance bottleneck at the single shared SSD. To show the performance impact of resource contention in a coupled architecture, we emulated an on-board consolidated storage unit by dedicating an NVMe SSD to an FPGA board through PCIe P2P. In this experiment, we implemented an example operator whose throughput is 100 MB/s and measured the end-to-end latency on a target VM with the increasing number of I/O-intensive VMs and their block I/O intensities (i.e., queue depth). Figure 6 shows the normalized end-to-end latency of the target near-storage processing on the coupled architecture. When two or more I/O-intensive VMs share the SSD, the near-storage processing latency becomes 2.1x slower than the single-VM execution case. When the VMs demand higher I/O performance by increasing the queue depth, the target VM and its near-storage processing suffer from the more severe resource contention.

3.3 Static SSD/FPGA Resource Allocation

Static resource allocation and scheduling for both computation and storage units will incur high hardware costs because they cannot handle the dynamic behavior of VM workloads efficiently. To motivate dynamic resource allocation and scheduling, we implemented two hardware operators on an FPGA and allowed each of them to serve two VMs using time multiplexing. Following the resource-sharing mechanisms proposed by prior FPGA virtualization studies [19, 26], we oversubscribed the hardware operators but statically assigned them to a specific set of VMs.

Figure 7 shows the latency cumulative distribution function of the four VM workloads with the static resource allocation strategy. In this experiment, we executed four VMs concurrently and generated VM workloads by following Poisson distribution with different expected request rates (λ). For the first two VMs (VM_1, VM_2), we generated near-storage processing workloads with the same execution time and wait time ($T_{exec} = T_{wait}, \lambda_A = \frac{1}{T_{exec} + T_{wait}} = \frac{1}{2T}$). For the other two VMs (VM_3, VM_4), we generated workloads with a longer period between near-storage processing invocations ($\lambda_B = \frac{1}{16T}$). The result demonstrates that the static operator allocation scheme cannot guarantee a target QoS with the skewed workloads. When the VM_1 and VM_2 share the same operator and invoke

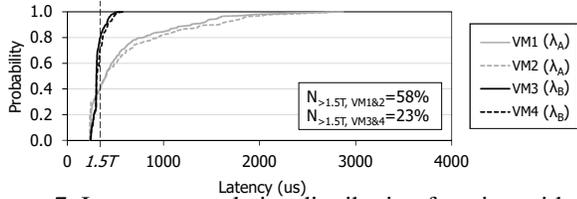


Figure 7: Latency cumulative distribution function with two oversubscribed hardware operators and four VMs.

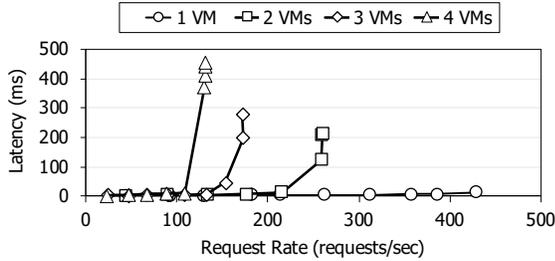


Figure 8: Operator latency with the increasing number of VMs and their request rate.

near-storage processing with a high request rate (λ_A), they severely suffer from a larger number of QoS violations ($> 1.5 \times T_{exec}$) than the other VM group (λ_B) due to the severe resource contention between the VMs.

The static resource allocation and scheduling mechanisms can lead to unacceptable QoS levels with the increasing number of VMs. Figure 8 shows the average latency of an encryption operator with varying request rates. If the total request rate does not exceed the maximum throughput of the operator, the operator can serve the requests within a reasonable latency bound. For example, when the four VMs submit requests at a rate of 50 requests per second each, the operator will show reasonable latency bound and the VMs will not suffer from the unexpected delay. However, when the total request rate exceeds the maximum throughput, the operator latency and the number of QoS violations increase quickly.

4 Design and Implementation

4.1 FlexCSV Architecture

In this work, we propose FlexCSV, a fast and cost-effective virtualization mechanism for computational storage devices. Its key idea is to implement *FCSV-Engine*, an FPGA card designed to maximize the performance and cost-effectiveness of computational storage virtualization. Figure 9 shows the *FCSV-Engine* architecture and its main hardware components. *FCSV-Engine* implements (1) a hardware-assisted virtualization layer based on PCIe SR-IOV, (2) a hardware-level direct device-orchestration mechanism, (3) an SSD-FPGA decoupled architecture, and (4) dynamic resource allocation through its operator renaming and partial reconfiguration support.

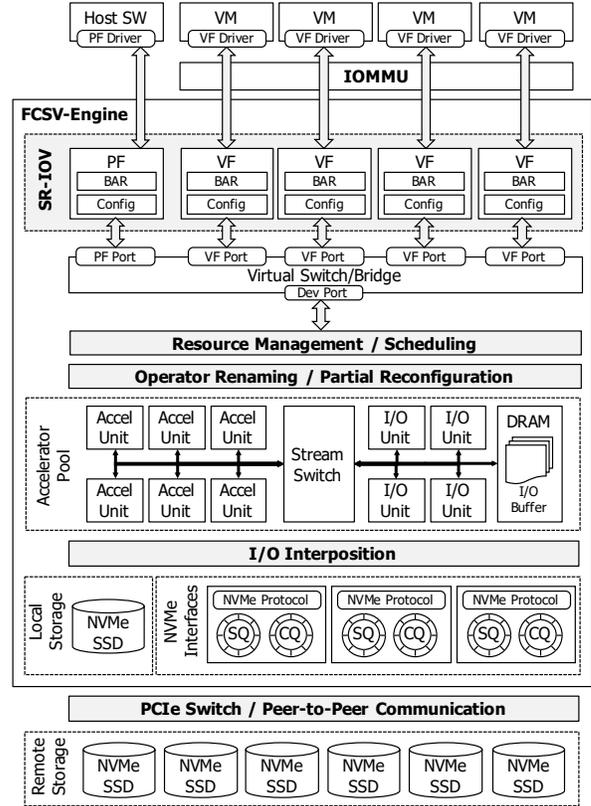


Figure 9: *FCSV-Engine* architecture.

4.2 Hardware-assisted Virtualization

4.2.1 SR-IOV and VM Isolation

To mitigate the software overhead in computational storage virtualization, *FCSV-Engine* offers hardware-assisted virtualization under a standard PCIe SR-IOV layer. By incorporating SR-IOV, *FCSV-Engine* can virtualize itself at the hardware level and each VF can be assigned exclusively to a VM for the direct access. Another advantage is that SR-IOV does not demand extra server CPU cores for polling guest I/O activities and indirect interrupt injections, which enables even more scalable and cost-effective server configurations than the conventional software-centric virtualization mechanisms (e.g., trap-and-emulate, sidecore). In this work, we utilize a single SR-IOV implementation at *FCSV-Engine* to virtualize both computation and storage units. This design choice minimizes the server costs for purchasing and operating SR-IOV supported computation and storage devices.

In addition, we utilize non-overlapping address translation to *FCSV-Engine*'s internal address space (i.e., PCIe-to-AXI address translation) to guarantee isolated execution of multiple VMs. *FCSV-Engine* allocates a disjoint set of memory regions and assigns different AXI address ranges for each VF so that near-storage processing requests from two different VMs do not interfere with each other. For example, we statically partition *FCSV-Engine*'s on-chip memory and off-chip

DRAM regions, and apply different offset values to MMIO requests from the VMs. Similarly, we allocate a different set of available interrupt vectors of FCSV-Engine for each VM. In this work, we allocate eight interrupt vectors per VM, and each vector is dedicated to a single completion queue of FlexCSV device driver running on a guest OS.

4.2.2 Device Interface

FCSV-Engine implements a multi-queue device interface and a doorbell mechanism to interact with guest OSEs. For this multi-queue device interface, we arrange FCSV-Engine’s PCIe BAR regions for doorbell registers of virtual FCSV-Engine instances (i.e., each VF). From the software side, FCSV-Engine’s device driver installed on a guest OS allocates multiple SQ/CQ pairs and initializes the doorbell registers mapped at FCSV-Engine’s BARs. FCSV-Engine’s device driver then delivers the guest physical addresses of the allocated queue pairs to FCSV-Engine. The number of queue pairs is dictated by FCSV-Engine’s BAR configuration and FPGA on-chip resource budgets. In this work, we create eight queue pairs per VM so that the same number of virtual CPUs can offload near-storage processing in parallel.

FCSV-Engine polls its on-chip memory space for doorbell registers using its host interfaces. To serve near-storage processing requests from multiple VMs concurrently, we instantiate multiple host interfaces and dedicate them to each VF. They get newly updated doorbell values by polling the doorbell register regions and utilize an internal DMA engine and an IOMMU to access a target SQ in guests’ memory space. By incorporating an IOMMU and its guest-to-host address translation mechanism, each virtual FCSV-Engine instance can safely access target queue pairs allocated in the guest memory space without software intervention. Alternatively, FCSV-Engine can manage a guest-to-host address mapping table in itself, but this design incurs significant VM memory overhead to store the translation tables for every VM.

4.3 Hardware-level Resource Orchestration

4.3.1 Near-storage Processing Command

To offload resource orchestration routines to FCSV-Engine, FlexCSV extends a standard NVMe protocol and defines a new command format for near-storage processing. FlexCSV’s NVMe-extended computational storage protocol minimizes software modifications to support near-storage processing in virtualized environments. Since major cloud providers are allowing NVMe storage devices to be used as primary storage for VMs, our NVMe-extended protocol can be easily applied in modern cloud and datacenter infrastructures.

Figure 10 shows a near-storage processing command structure. First, `op_chain` specifies which operators should be activated and the target stream order of the activated operators. For this, every operator type implemented on an FPGA

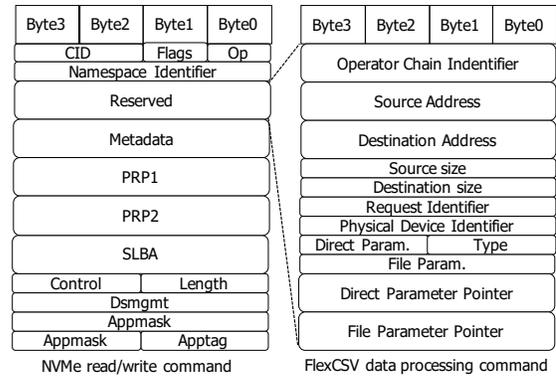


Figure 10: Near-storage processing command in FlexCSV.

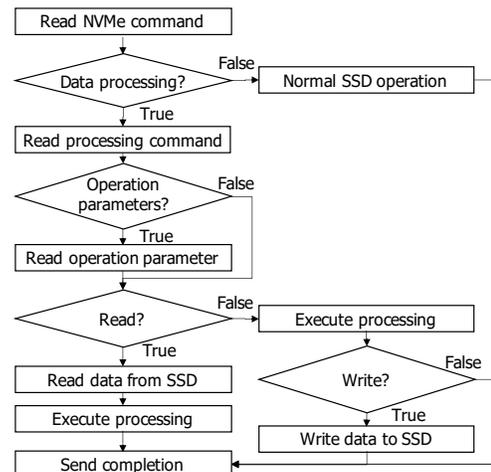


Figure 11: Control flow in FCSV-Engine.

has a unique identifier. `src_addr/size` and `dst_addr/size` represent the addresses and sizes of the source and destination in the FPGA’s DRAM space. Additionally, if an operator needs parameters to process, a user can carry the parameters either directly or indirectly with `direct_param` or `file_param` fields. A user can also manage dependency between near-storage processing commands by manipulating a `request_id` field. If a request contains the same `rid` as in the previous requests, the current request cannot be issued before the earlier requests finish their near-storage processing. `type` determines whether a current request involves storage access or not.

4.3.2 Resource Orchestration

FCSV-Engine involves a resource orchestration mechanism to manage both computation and storage resources. To orchestrate two different hardware units without frequent software-hardware crossings, FCSV-Engine schedules user-requested computation and storage operations at the hardware level. Figure 11 shows the hardware-level scheduling mechanism. First, FCSV-Engine identifies a newly issued command by polling submission doorbell registers. If the command re-

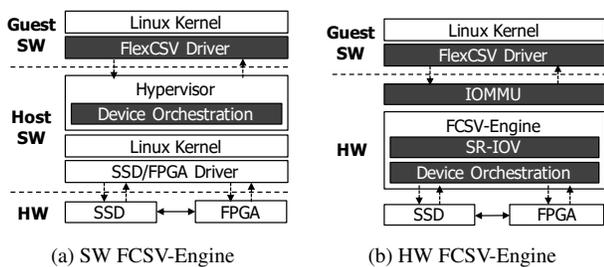


Figure 12: FCSV-Engine implemented in SW and HW.

quires near-storage processing, FCSV-Engine reads the data processing command and saves its contexts (e.g., source/destination FPGA DRAM addresses). After that, FCSV-Engine manipulates the DMA buffer addresses of the received NVMe command so that the SSD can transfer the user-requested data to/from the FPGA’s DRAM. FCSV-Engine also enforces a correct order of computation and storage operations depending on the direction of user-requested data movements (i.e., read or write) and target operators.

FCSV-Engine executes user-requested stream data processing without software orchestration. In contrast to the existing static routing mechanisms, FCSV-Engine adopts a dynamic routing mechanism between hardware operators. For this, FCSV-Engine’s control logic determines the order and routing path dynamically when it executes near-storage processing. FCSV-Engine parses the `op_chain` field of a data processing command and reserves the shortest routing path by manipulating the `dest` configuration registers of the user-requested operators. The operators then manipulate the `dest` field of the output stream data and FCSV-Engine’s internal crossbar switch redirects the incoming stream to the correct next operator. If the requested operators are used and their paths are already reserved, FCSV-Engine stalls their executions until the earlier requests finish their near-storage processing.

4.3.3 Software FCSV-Engine

FCSV-Engine’s resource orchestration mechanism can be implemented at the hypervisor level, but it still suffers from frequent software-hardware layer crossings to orchestrate two different hardware units. Figure 12 illustrates the software and hardware FCSV-Engine implementations. In the software implementation, a guest OS can leverage an NVMe-extended near-storage processing protocol and thus reduce the number of guest-host layer transitions. It also allows a device to directly transfer data to another device’s internal memory by manipulating the DMA buffer addresses to the target device memory addresses. However, this design suffers from the inevitable hypervisor and host OS overhead due to the indirect orchestration for SSD and FPGA devices and a large number of software-hardware layer transitions.

Because of the indirect resource orchestration routines through the host software (e.g., MMIO, interrupt), the soft-

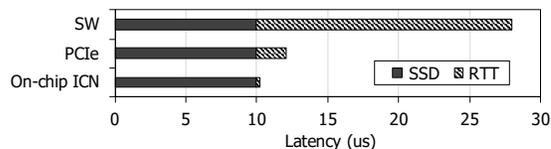


Figure 13: Storage latency and round-trip time through SW, PCIe, and on-chip ICN.

ware FCSV-Engine implementation fails to achieve the full potential of near-storage processing [21]. To profile the performance overhead of the software FCSV-Engine implementation, we measured the SSD access latency through (1) the host software, (2) PCIe P2P, and (3) on-chip interconnect (ICN). Figure 13 shows their round-trip time for access to an Intel Optane SSD. The result indicates that the major performance bottleneck is moved to the software stacks when the host software manages a device operation. Such performance overhead will get worse when we exploit near-storage processing between faster computation and storage units. On the other hand, the round-trip time of the PCIe P2P and on-chip ICN is still much faster than the software MMIO and interrupt mechanisms.

4.4 SSD-FPGA Decoupled Architecture

In this work, we introduce a decoupled computational storage architecture for scalable near-storage processing with multiple PCIe-attached SSDs. To allow multiple NVMe SSDs to combine with FCSV-Engine through PCIe P2P, the host software remaps their queue pairs onto FCSV-Engine’s BAR regions. By doing so, the decoupled storage units can seamlessly exchange NVMe commands and their completions with FCSV-Engine. The SSDs are unaware of being interacting with FCSV-Engine, but an external PCIe switch delivers their PCIe read and write transactions to FCSV-Engine directly. In addition, FlexCSV can nicely scale with a large number of PCIe-attached SSDs leveraging its large on-chip memory space. As a result, FlexCSV can mitigate the performance bottleneck at a single FPGA or SSD by flexibly combining PCIe-attached computation and storage devices in the same server.

Moreover, FCSV-Engine implements PCIe message arbitration and transaction modules to encapsulate local NVMe requests (e.g., NVMe doorbell write) with PCIe transactions and allow multiple storage interfaces to share a single PCIe/DMA IP core. In this arbitration module, FCSV-Engine leverages PCIe transaction queues for each VM and adopts a round-robin algorithm to serve PCIe access from multiple VMs. The PCIe transaction module translates an internal AXI address to an associated host physical address (e.g., AXI-to-PCIe address translation). For this address translation, the PCIe/DMA IP core manages an AXI-to-PCIe address mapping table. The current Xilinx-provided PCIe/DMA IP core supports up to six mapping entries, and this design point can be a serial

	Critical Path		FPGA Resource Usage		
	Submission	Completion	LUT	FF	BRAM
Static	656 ns	72 ns	20370	6702	11
Renaming	728 ns	96 ns	20693	6892	11

Table 1: Renaming overhead analysis.

point when it handles concurrent near-storage processing from many VMs.

4.5 Dynamic Resource Allocation

4.5.1 Operator Renaming

To maximize the hardware resource efficiency, FCSV-Engine implements dynamic resource allocation through its operator renaming support. FCSV-Engine implements a shared operator pool and dynamically maps user-requested operations onto available physical operators. In this way, FCSV-Engine can quickly capture the dynamic behavior of VM workloads and thus reduce QoS violations significantly. Our current resource scheduling mechanism is similar to a resource availability-based FCFS algorithm because we focus more on cost-effectiveness and resource utilization of computation and storage units. However, we can also improve storage fairness and performance by adopting more fairness-oriented scheduling methods [16, 36].

In the renaming stage, every operator request from VMs is mapped onto physical operators via an operator renaming table. The operator renaming table manages the availability of physically implemented operators. FCSV-Engine looks up the renaming table to find and allocate available physical instances of user-requested operators. If it succeeds in allocating the physical operators, the scheduler executes near-storage processing and manipulates the operator renaming table to record the resource allocation status. When the requested near-storage processing finishes, the scheduler collects the completions from all the activated operators and deallocates the recorded resources by manipulating the renaming table.

We measured the area and performance overhead when the operator renaming mechanism is implemented in FCSV-Engine’s scheduler module. We first implemented the scheduler without the renaming capabilities in which virtual operators have a one-to-one mapping with physical operators, and implemented operator renaming logic on top of it. In this work, our operator renaming logic can remap a virtual operator onto four physical operator instances. Table 1 shows the increase in the area and critical path for our operator renaming logic. The area overhead is 2%–4% and the total critical path overhead is around 100 ns, which is negligible compared to the original scheduler area and operator delay.

4.5.2 Operator Partial Reconfiguration

Partial reconfiguration (PR) support for FPGA operators further improves the hardware utilization by capturing the dy-

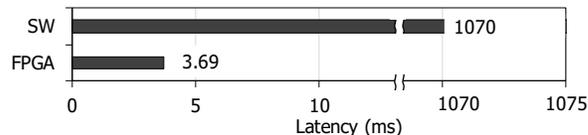


Figure 14: Partial reconfiguration latency with the host software and an FPGA.

	CLB LUTs	CLB Registers	Block RAM Tiles	Clock speed
FCSV-Engine	313,069 (18%)	487,333 (14%)	441 (16%)	250 MHz
PCIe / SR-IOV	46,970	51,487	61	
DRAM	18,592	20,817	26	
Host Interface (4 VMs)	74,886	124,761	266	
Storage Interface	22,927	39,403	39	
Scheduler	2,507	3,050	4	
Interconnect	115,073	203,827	0	
Others	32,114	43,988	45	

Table 2: FCSV-Engine FPGA resource utilization.

amic behavior of VM workloads. If there exist enough operator slots to serve all requested near-storage processing workloads, their target operators can be assigned to available slots in FCSV-Engine’s operator pool. However, if the demand from near-storage processing workloads exceeds the maximum number of operator slots, they can be partially reconfigured to serve the current user requests. As a result, such a dynamic resource allocation mechanism enables the FPGA to support more operators than its physical resource limit.

Figure 14 shows the latency of partially reconfiguring the FPGA operators from the host software and from the FPGA itself. Reconfiguration from the host software incurs long latency because the software has to send the bitstream from the host DRAM to the FPGA. Alternatively, to achieve agile operator reconfiguration, we can store the partial bitstreams in the FPGA DRAM and implement a PR controller to allow the FPGA to reconfigure its operator slots. In this way, the PR latency is reduced by 99.7% and the scheduler in FCSV-Engine can dynamically generate the operators on demand with low overhead.

5 Evaluation

5.1 Experimental Setup

To evaluate FlexCSV, we implemented our FCSV-Engine prototype on a Xilinx Alveo U250 board and installed its SW support on the Linux KVM/QEMU virtualization stack. Our FCSV-Engine prototype provides eight stream-based operators with the partial reconfiguration support and allows the operators to be shared by four VMs through the hardware-assisted virtualization and resource allocation mechanisms.

Operators	LUTs	Registers	BRAMs	Performance
Encryption (E) [28]	38,031	19,246	168	211 MB/s
Decryption (D) [28]	37,655	19,126	168	212 MB/s
Hash (H) [37]	50,620	13,541	0	285 MB/s
Aggregate (A) [35]	2,292	1,539	2	4.70 GB/s
Filter (F) [35]	41,823	5,428	116	278 MB/s
Grep (G) [15]	37,288	5,647	6	426 MB/s
KNN (K) [32]	14,052	4,091	8	4.22 GB/s
Bitmap (B) [33]	63,570	5,588	31	4.24 GB/s

Table 3: Operators’ FPGA resource utilization.

To enable FCSV-Engine to interact with CPUs and PCIe-attached NVMe SSDs, we utilized Xilinx-provided PCIe and DMA engine implementations (PCIe Gen3 4-lane, 4 GB/s per direction) [9] and configured VFs to virtualize FCSV-Engine itself. For intermediate data buffers between computation and storage units, FCSV-Engine leverages its on-board DDR4 DRAM and on-chip AXI-stream FIFO queues.

Table 2 shows FCSV-Engine’s FPGA resource utilization using Xilinx Vivado and HLS (v2019.2). The on-chip interconnect logic consumes the major portion of FPGA LUTs and registers as it connects all the VFs (i.e., host interfaces for each VM) and operators through all-to-all crossbars. On the other hand, the host and storage interfaces consume many on-chip memory tiles for their device register regions of the NVMe-extended protocol (e.g., doorbell registers) and for NVMe I/O queue regions to orchestrate PCIe-attached NVMe SSDs. Note that the FPGA has enough remaining resources to add more operator slots for near-storage processing.

Our software-hardware full-system prototype is built on a host server with two Intel’s Xeon Gold 5118 CPUs, each with 12 physical cores running at 2.3 GHz, and 256-GB DDR4 DRAM (Supermicro SuperServer 4029GP-TRT2). The host server machine is equipped with four Intel Optane 900P NVMe SSDs and connects them to FCSV-Engine through PCIe Gen3 4-lane interconnects (4 GB/s per direction). The Optane SSD can offer up to 550k IOPS in random-read and 500k IOPS in random-write with 10 μ s latency. For software support, we installed a Ubuntu 18.04 OS and Linux kernel (version 5.3) on VMs and implemented custom FCSV-Engine’s device driver.

To generate various near-storage processing scenarios, we implemented eight representative computational storage operations from the previous FPGA acceleration and near-storage processing studies [15, 28, 32, 33, 35, 37]. We utilized the Xilinx HLS tool (v2019.2) to generate their hardware operators. The implemented hardware operators perform stream operations through 512-bit input and output data links (16 GB/s interconnect bandwidth with 250-MHz clock frequency). In this work, we dedicated 4x more on-chip bandwidth than the configured PCIe link capability (4 GB/s per direction) to avoid the bottleneck at the on-chip crossbars and to obtain the full potential of near-storage processing in multi-VM workloads. However, the proper data stream width may vary depending

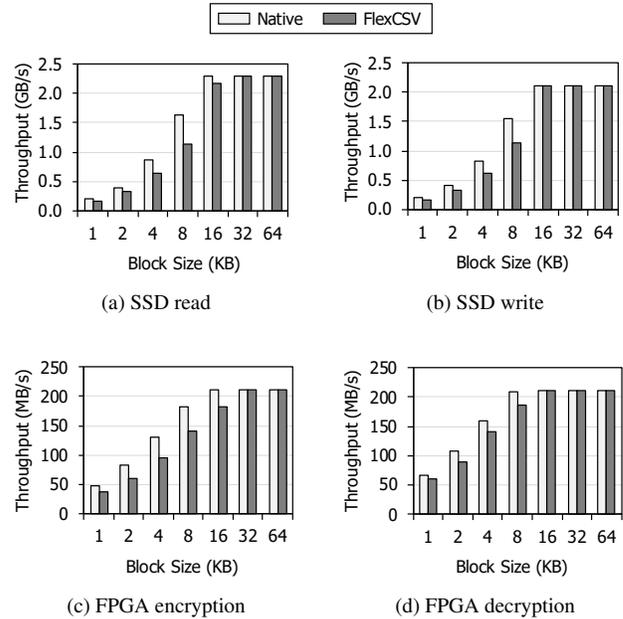


Figure 15: SSD and FPGA operation performance with different virtualization mechanisms.

on the target operators and resource budgets.

The FPGA operators have a broad spectrum of resource utilization and data-processing throughput based on their algorithm complexity and synthesis strategies. Table 3 shows the benchmarks and their FPGA implementation results. The Blowfish data encryption and decryption operators consume a large number of registers, but demonstrate the lowest data processing performance among the benchmarks. On the other hand, the aggregate, K-nearest neighbors (KNN), and bitmap operators show the highest ideal performance due to their algorithmic simplicity.

5.2 Device Virtualization Performance

In this experiment, we measured the performance of the hardware-assisted virtualization mechanism and compared it with the native device performance. For this evaluation, we installed the SSD and FPGA devices through PCIe and implemented the Blowfish encryption and decryption operators [28] on the FPGA. To fairly compare the performance for both device types, we measured the performance of the computation and storage units separately. To measure the storage performance, we ran flexible I/O tester (FIO) [2] in both native and FlexCSV virtualization environments with an increasing data block size. In contrast, we leveraged the NVMe-extended protocol to measure the FPGA virtualization performance without invoking storage operations. Similar to the storage benchmarks, the FPGA benchmarks measured the end-to-end performance in both native and FlexCSV environments with an increasing data block size.

The experimental results show that our FCSV-Engine prototype delivers near-native NVMe SSD and FPGA performance.

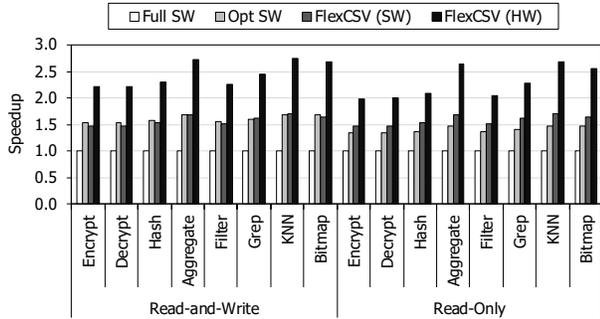


Figure 16: Speedup comparison in various virtualized environments.

Figure 15 shows the native and virtualized SSD and FPGA acceleration performance. Due to the hardware-assisted virtualization mechanism (including SR-IOV), FCSV-Engine can achieve the near-native performance when utilizing both SSD and FPGA devices in virtualized environments. Moreover, as the block size increases, the native and virtualized executions of the SSD and FPGA operators demonstrate the ideal bare-metal throughput (explained in Section 5.1 and Table 3). The increasing data block size further mitigates the software overhead by merging multiple near-storage processing requests using NVMe’s scatter-gather list (SGL) support.

5.3 Near-Storage Processing Performance

In this experiment, we measured the performance of the near-storage processing benchmarks through existing software stacks and FlexCSV virtualization mechanisms. For this evaluation, we generated a 4-GB dataset as an input file of near-storage processing, and each benchmark running on a guest OS divides the dataset into multiple 4-KB blocks and iterates them to cover the total dataset size. To compare the speedup values over the existing software-centric mechanisms, we also executed the same near-storage processing benchmarks on the paravirtualization schemes with and without SR-IOV support at the SSD side (described in Figure 4). In addition, we measured the performance of software FCSV-Engine (described in Figure 12) to highlight the benefits of FlexCSV’s hardware-level resource orchestration.

Each near-storage processing benchmark listed in Table 3 is executed in two VM workload scenarios. The first scenario reads a data block from the SSD directly, performs data processing using the FPGA’s stream operator, and writes its output data to the SSD (read-and-write). The other scenario, on the other hand, follows the same routines for every data block, but does not write back the output data for further near-storage processing (read-only).

The experimental results show that FlexCSV can achieve 2.1x (geomean) faster near-storage processing in virtualized environments over the software virtualization mechanisms. Figure 16 shows the speedup values compared to the full software virtualization mechanism. In contrast to the software-

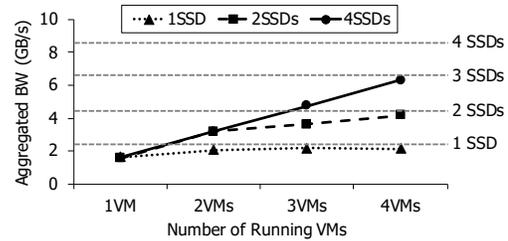


Figure 17: The aggregated bandwidth with a different number of VMs and SSDs.

based virtualization mechanisms, FlexCSV allows the VM to access the underlying devices and orchestrates the data movements between the virtualized units without any software intervention. The encryption and decryption benchmarks obtain 2.2x speedup (33 MB/s→72 MB/s) in the read-and-write case and 2x speedup (48 MB/s→95 MB/s) in the read-only case. The KNN benchmark, which shows the highest speedup values among the benchmarks, achieves 2.8x speedup over the full software virtualization (38 MB/s→105MB/s in the read-and-write case). Compared to the software FCSV-Engine implementation, the hardware FCSV-Engine implementation achieves 1.4x speedup on average.

5.4 Multi-SSD Performance

In this work, we evaluated FlexCSV’s decoupled architecture by executing an I/O-intensive VM workload with an increasing number of VMs and PCIe-attached SSDs. The result is shown in Figure 17. In this experiment, the VMs utilize about 2 GB/s storage bandwidth, similar to the maximum bandwidth of a single SSD. A single SSD can meet the performance requirements of a single VM workload as there is no storage interference. However, when we utilize a single SSD and run two or more VMs, the aggregate bandwidth of all the VMs is limited by a single SSD. So, the tightly-coupled architectures suffer from such bandwidth imbalances and unexpected delays as the available storage bandwidth cannot be scaled easily. However, with the FPGA and SSD decoupled, the required bandwidth of multiple VMs can be met by adding more SSDs. In this work, FCSV-Engine supports attaching up to four SSDs to a single FPGA and achieves the scalable performance with the increasing number of VMs and SSDs.

5.5 Dynamic Resource Scheduling

In this experiment, we evaluated the effectiveness of FCSV-Engine’s dynamic resource allocation and scheduling mechanisms. For this evaluation, we ran four VMs concurrently and generated the VM workload by following Poisson distribution with diverse expected request rates ($\lambda = \frac{1}{T} = \frac{1}{T_{exec} + T_{wait}}$). We ran four VMs and grouped the VMs into two groups (A, B) that have different request rates (T_{wait}/T_{exec}). After that,

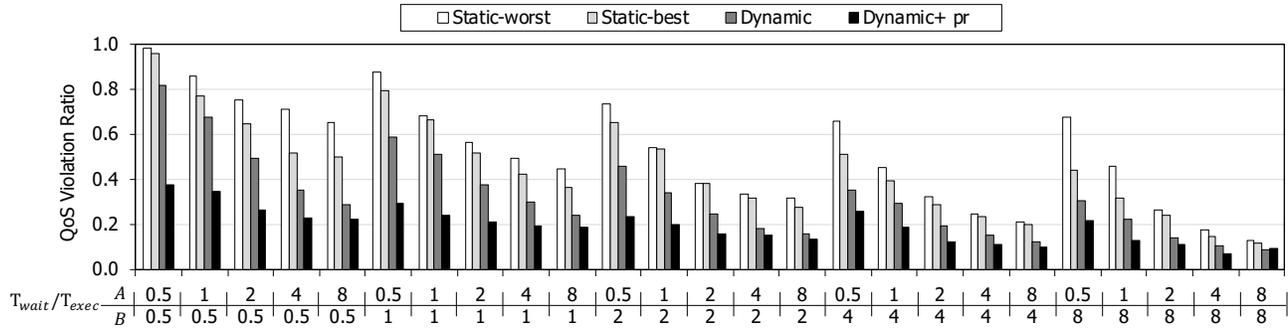


Figure 18: QoS violation ratios with different request rates from multiple VMs.

we compared their QoS violation ratios with four different resource scheduling strategies: (1) static-worst, (2) static-best, (3) dynamic, and (4) dynamic + partial reconfiguration (dynamic+pr). The static-worst, static-best, and dynamic scheduling methods utilize two physical operators, and dynamic+pr can support up to four physical operators through partial reconfiguration. The VMs in the experiment perform an encryption operation with near-storage processing capabilities.

Figure 18 shows the QoS violation ratios ($> 1.5 \times T_{exec}$) with diverse request rate combinations of the two VM groups. Except when the request rate between the two groups is the same, the static-best scheduling methods achieve lower the QoS violation ratios compared to the static-worst method. Also, the dynamic scheduling significantly lowers the QoS violation ratios over the static-worst and static-best strategies for all the workload scenarios. When both VM groups invoke the operators frequently (e.g., $[A=0.5, B=0.5]$), FCSV-Engine can increase the number of available operators using its partial reconfiguration support and further reduce the QoS violation ratios. If the request rate is low (e.g., $[A=4, B=8]$), the dynamic scheduling can drop QoS violations, but increasing the number of operators has a minor impact.

6 Related Work

Near-storage processing. Recent near-storage processing studies have proposed flexible FPGA overlay architectures and improved the usability of computational storage devices [33, 35]. Their FPGA overlay architectures can implement user-specified operators and activate them selectively through the crossbar switches and control logic. Also, their custom software stacks provide abstraction layers to hide the complexity of the underlying FPGA implementations.

NVMe SSD virtualization. NVMe virtualization becomes one of the most critical components in cloud environments to meet the performance demand from modern server workloads. For example, Amazon Web Services (AWS) accelerates I/O virtualization through dedicated hardware components. To make full use of its parallel and high-performance storage protocol, storage performance development kit (SPDK) vhost-

nvme extends the SPDK library to provide virtual NVMe controllers to QEMU-based VMs [38]. Similarly, another implementation provides a mediated passthrough mechanism in kernel space with an active polling mode [29]. In addition, current hardware-assisted virtualization studies demonstrate offloading NVMe virtualization stacks to a programmable FPGA or SmartNIC device [22, 25].

FPGA virtualization. The existing FPGA virtualization studies introduce abstractions for FPGA logic cell and interconnection components [19, 20, 40]. First, user logic is encapsulated in flexible operators to be dynamically scaled and remapped to the physical fabric. Second, the host software manages the mapping between operators and physical FPGAs. To enable flexible mapping, high-level operators encapsulate information to enable the host software to generate new FPGA implementations on demand. Moreover, the host software maintains a registry to hide the latency of partial reconfiguration for dynamic scalability.

7 Conclusion

In this work, we propose a fast, flexible, and cost-effective mechanism to virtualize computational storage devices. The key idea is to use FCSV-Engine, an FPGA card designed to maximize the performance and cost-effectiveness of computational storage virtualization. It achieves high virtualization performance by applying hardware-assisted virtualization and resource orchestration. In addition, it achieves high cost-effectiveness by dynamically constructing many virtual computational storage devices and scheduling their near-storage processing at the hardware level.

Acknowledgments

This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1901-12. We also appreciate the support from Automation and Systems Research Institute (ASRI) and Inter-university Semiconductor Research Center (ISRC) at Seoul National University.

References

- [1] Amazon EC2 F1 Instances. <https://aws.amazon.com/ec2/instance-types/f1/>.
- [2] Flexible I/O Tester. <https://github.com/axboe/fio>.
- [3] Intel Solid State Drives. <https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives.html>.
- [4] Linux KVM. https://www.linux-kvm.org/page/Main_Page.
- [5] QEMU. <https://www.qemu.org/>.
- [6] Samsung SmartSSD Computational Storage. <https://samsungsemiconductor-us.com/smartssd/>.
- [7] Xilinx Alveo U250 FPGA. <https://www.xilinx.com/products/boards-and-kits/alveo/u250.html>.
- [8] Xilinx FPGAs & 3D ICs. <https://www.xilinx.com/products/silicon-devices/fpga.html>.
- [9] Xilinx QDMA Subsystem for PCI Express. <https://www.xilinx.com/products/intellectual-property/pcie-qdma.html>.
- [10] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs. In *2017 USENIX Annual Technical Conference (ATC 17)*, pages 167–179, 2017.
- [11] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient Software Packet Processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990, 2020.
- [12] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.
- [13] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.
- [14] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale DNN processor for real-time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2018.
- [15] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moon-sang Kwon, Chanho Yoon, Sangyeun Cho, et al. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 153–165. IEEE, 2016.
- [16] Ajay Gulati, Irfan Ahmad, Carl A Waldspurger, et al. Parda: Proportional allocation of resources for distributed storage access. In *FAST*, volume 9, pages 85–98, 2009.
- [17] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, et al. Bluedbm: An appliance for big data analytics. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13. IEEE, 2015.
- [18] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42. IEEE, 2018.
- [19] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J Rossbach. Sharing, Protection and Compatibility for Reconfigurable Fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 107–127, 2018.
- [20] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do OS abstractions make sense on FPGAs? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 991–1010, 2020.
- [21] Dongup Kwon, Jaehyung Ahn, Dongju Chae, Mohammadamin Ajdari, Jaewon Lee, Suheon Bae, Youngsok Kim, and Jangwoo Kim. DCS-ctrl: a fast and flexible device-control mechanism for device-centric server architecture. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 491–504. IEEE, 2018.
- [22] Dongup Kwon, Junehyuk Boo, Dongryeong Kim, and Jangwoo Kim. FVM: FPGA-assisted Virtual Device

- Emulation for Fast, Scalable, and Flexible Storage Virtualization. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 955–971, 2020.
- [23] Alex Landau, Muli Ben-Yehuda, and Abel Gordon. SplitX: Split Guest/Hypervisor Execution on Multi-Core. In *WIOV*, 2011.
- [24] Joo Hwan Lee, Hui Zhang, Veronica Lagrange, Praveen Krishnamoorthy, Xiaodong Zhao, and Yang Seok Ki. SmartSSD: FPGA Accelerated Near-Storage Data Analytics on SSD. *IEEE Computer Architecture Letters*, 19(2):110–113, 2020.
- [25] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan RK Ports, Irene Zhang, Ricardo Bianchini, Haryadi S Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 591–605, 2020.
- [26] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. A hypervisor for shared-memory fpga platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 827–844, 2020.
- [27] Vikram Sharma Mailthody, Zaid Qureshi, Weixin Liang, Ziyang Feng, Simon Garcia De Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wen-mei Hwu. DeepStore: in-storage acceleration for intelligent queries. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 224–238, 2019.
- [28] Shraphalya B Nalawade and Dhanashri H Gawali. Design and implementation of blowfish algorithm using reconfigurable platform. In *2017 International Conference on Recent Innovations in Signal processing and Embedded Systems (RISE)*, pages 479–484. IEEE, 2017.
- [29] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. MDev-NVMe: A NVMe storage virtualization solution with mediated pass-through. In *2018 USENIX Annual Technical Conference (ATC 18)*, pages 665–676, 2018.
- [30] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy 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.
- [31] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE, 2014.
- [32] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia applications. In *Proceedings of 24th Conference on Very Large Databases*, pages 62–73. Citeseer, 1998.
- [33] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive. In *2019 USENIX Annual Technical Conference (ATC 19)*, pages 379–394, 2019.
- [34] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.
- [35] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. Accessible Near-Storage Computing with FPGAs. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–12, 2020.
- [36] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. Flin: Enabling fairness and enhancing performance in modern nvme solid state drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 397–410. IEEE, 2018.
- [37] Kurt K Ting, Steve CL Yuen, Kin-Hong Lee, and Philip HW Leong. An fpga based sha-256 processor. In *International Conference on Field Programmable Logic and Applications*, pages 577–585. Springer, 2002.
- [38] Ziye Yang, Changpeng Liu, Yanbo Zhou, Xiaodong Liu, and Gang Cao. SPDK Vhost-NVMe: Accelerating I/Os in Virtual Machines on NVMe SSDs via User Space Vhost Target. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pages 67–76. IEEE, 2018.
- [39] Joohyeong Yoon, Won Seob Jeong, and Won Woo Ro. Check-in: in-storage checkpointing for key-value store system leveraging flash-based SSDs. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 693–706. IEEE, 2020.

- [40] Yue Zha and Jing Li. Virtualizing FPGAs in the Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 845–858, 2020.
- [41] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, et al. FlashShare: Punching through server storage stack from kernel to firmware for ultra-low latency SSDs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 477–492, 2018.

Fair Scheduling for AVX2 and AVX-512 Workloads

Mathias Gottschlag
Karlsruhe Institute of Technology
Yussuf Khalil
Karlsruhe Institute of Technology

Philipp Machauer
Karlsruhe Institute of Technology
Frank Bellosa
Karlsruhe Institute of Technology

Abstract

CPU schedulers such as the Linux Completely Fair Scheduler try to allocate equal shares of the CPU performance to tasks of equal priority by allocating equal CPU time as a technique to improve quality of service for individual tasks. Recently, CPUs have, however, become power-limited to the point where different subsets of the instruction set allow for different operating frequencies depending on the complexity of the instructions. In particular, Intel CPUs with support for AVX2 and AVX-512 instructions often reduce their frequency when these 256-bit and 512-bit SIMD instructions are used in order to prevent excessive power consumption. This frequency reduction often impacts other less power-intensive processes, in which case equal allocation of CPU time results in unequal performance and a substantial lack of performance isolation.

We describe a modification to existing schedulers to restore fairness for workloads involving tasks which execute complex power-intensive instructions. In particular, we present a technique to identify AVX2/AVX-512 tasks responsible for frequency reduction, and we modify CPU time accounting to increase the priority of other tasks slowed down by these AVX2/AVX-512 tasks. Whereas previously non-AVX applications running in parallel to AVX-512 applications were slowed down by 24.9% on average, our prototype reduces the performance difference between non-AVX tasks and AVX-512 tasks in such scenarios to 5.4% on average, with a similar improvement for workloads involving AVX2 applications.

1 Introduction

One common requirement for schedulers is to provide an acceptable quality of service to the tasks in the system [7]. Fair schedulers evenly share CPU performance among the tasks or groups of tasks in the system to achieve good quality of service for all tasks [15]. In the past, it was commonly assumed that even shares of the CPU time result in even shares of CPU performance. The Linux Completely Fair Scheduler [20], for

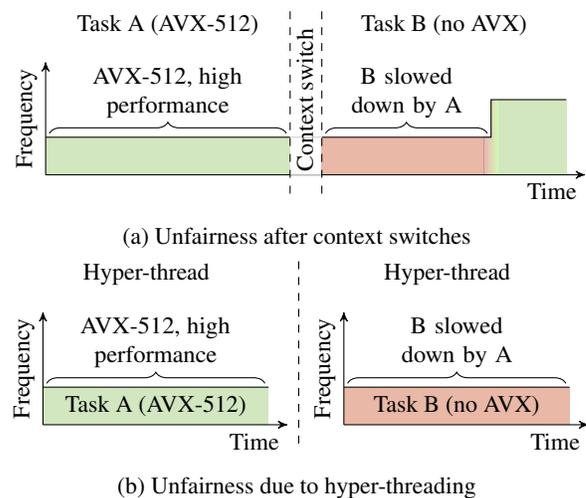


Figure 1: With AVX2 and AVX-512, equal CPU time shares do not translate into fair shares of CPU performance. AVX2 and AVX-512 cause the CPU core to reduce its frequency which can affect other non-AVX tasks. In these situations, our prototype would increase the priority of task B to achieve improved fairness. In this figure, the two frequencies represent the non-AVX and AVX-512 turbo levels.

example, always executes the task with the lowest CPU time used (scaled by priority) to allocate equal shares of CPU time to individual tasks.

In systems with dynamic voltage and frequency scaling (DVFS), the assumption that equal CPU time results in equal performance does not hold. Reduced frequencies affect the performance of applications differently depending on how memory-limited the applications are, and techniques have been developed to include this effect when distributing CPU time [14]. Such techniques have not found their way into common operating systems, though, most likely because fairness mostly matters when system utilization is high, in which case operating systems rarely reduce the CPU frequency.

Recent CPUs, however, have started to use autonomous DVFS outside of the control of the operating system to prevent excessive power consumption. In the last years, transistor scaling has increased power density, leading to a situation where the performance of recent CPUs is largely limited by their power consumption [28]. One common technique to improve CPU performance in this power-limited design regime is to use additional transistors to implement specialized accelerators which remain inactive most of the time. CPU cores can operate at high frequencies while the accelerators are inactive, whereas use of the accelerators requires a temporary frequency reduction to prevent excessive power draw. This adaptation of the CPU frequency to meet thermal and power supply constraints increases individual cores' frequencies to always utilize their whole power budget but also decreases frequencies if the cores risk overheating or instability, similar to techniques such as Turbo Boost [1] which increases the frequency of the whole processor when some cores are idle.

Recent Intel CPUs provide an example for such behavior where low-power code is executed at increased frequencies. For some workloads, these CPUs provide a substantial performance advantage over previous CPU generations due to the introduction of the AVX2 and AVX-512 SIMD instruction sets [6] which support operations on up to 512-bit vector registers. The large difference in power consumption between AVX2/AVX-512 instructions and other instructions, however, requires individual CPU cores to reduce their frequency while they execute AVX2 and AVX-512 code, whereas other less power-intensive code is executed at higher frequencies [23]. As an example, the Intel Xeon Gold 6130 server CPU executes non-AVX code at an all-core turbo frequency of 2.8 GHz, whereas AVX2 and AVX-512 code is executed at 2.4 GHz and 1.9 GHz, respectively [4].

If only AVX2 or AVX-512 code would be selectively slowed down, no problem for fair schedulers would arise. Only tasks *choosing* to execute AVX2 or AVX-512 code would be slowed down and the increased throughput of these instructions would more than compensate the frequency reduction. However, the impact of the lower frequency is not limited to the AVX2 and AVX-512 code responsible for the reduction [10]. In two scenarios, the frequency reduction also affects other potentially unrelated tasks:

1. Context switches: When the CPU switches from an AVX2 or AVX-512 task to a non-AVX task, the CPU waits for a fixed timeout before restoring the standard frequency [5]. Such a delay is sensible as it limits worst-case overhead due to frequent clock changes. The following task is consequently slowed down as well.
2. Hyper-threading: When one hyper-thread executes either AVX2 or AVX-512 code, the whole physical core needs to reduce its frequency, so any code running on the other hyper-thread is affected as well even if it does not execute AVX2 or AVX-512 instructions [10]. This effect

has shown to be particularly prominent, with workloads often being slowed down by more than 15% when executed alongside an AVX-512 application on Linux [10].

As shown in Figure 1, when contemporary fair schedulers allocate the same share of CPU time to AVX2/AVX-512 tasks which cause a frequency reduction as to other tasks which are also affected by the reduced frequencies, the latter would receive a substantially reduced share of the system's performance. This unfairness and the resulting performance reduction is particularly problematic for tasks with soft real-time requirements and for multi-tenant systems where individual users commonly pay to receive a specific share of the CPU time. While these effects are currently only observed on systems with recent Intel CPUs, we expect future power-limited systems to show similar behavior as outlined above.

In this paper, we show that for these systems it is necessary to rethink our notions of scheduling fairness. In particular, we show that equal CPU time, minimal effects on quality of service, and fair distribution of CPU performance are mutually exclusive goals. We do so by describing a scheduler that is commonly able to achieve the latter when some tasks – called *victim tasks* in the following – are negatively affected by a frequency reduction caused by other tasks.

We present a concrete implementation of this design for Intel systems and AVX2 and AVX-512 instructions. Our approach mitigates the performance impact of frequency reduction via modified CPU time accounting where the CPU time of victim tasks is scaled according to the decrease in CPU frequency (Sections 3). We show how victim tasks can often be recognized by the lack of characteristic register accesses (Section 4). Variable turbo frequencies complicate calculating the actually experienced frequency reduction, so we show how on Intel systems the average frequency reduction during a scheduling time slice can be calculated using only two configurable performance counters and a cycle counter (Section 5). As the load balancing mechanism of the Linux CFS scheduler prevents achieving fairness, we describe an implementation of our design based on the Linux MuQSS scheduler modified to use the CFS scheduling algorithm (Section 6). Our evaluation based on a range of real-world applications shows that our prototype has close to zero runtime overhead, yet is able to improve the fairness in workloads with AVX2 and AVX-512, reducing the performance difference between two applications from 24.9% to 5.4% if one of the applications uses AVX-512 (Section 7). Finally, we discuss different fairness criteria, showing how our approach can be modified to achieve even stronger performance isolation (Section 8), as well as the limitations of our approach (Section 9).

2 AVX Frequency Reduction

To mitigate the unfairness caused by power-intensive instructions, we first need to know which instructions cause the

CPU to reduce its frequency and by how much the frequency is reduced. Our prototype targets the AVX2 and AVX-512 instruction sets. Both are categorized by Intel into “heavy” and “light” instructions, where the former consist of all floating point instructions as well as multiplication instructions, whereas the latter consist of all other instructions [5]. In the default CPU configuration at the maximum non-AVX frequency, all light and heavy instructions have the potential to draw excessive current, which causes voltage drops and system instability. During execution of all these instructions, the CPU therefore increases its voltage to allow for increased currents [8].

Not all 256-bit and 512-bit instructions, however, also cause a frequency change. CPUs with support for AVX-512 provide three different frequency ranges named “non-AVX frequencies”, “AVX2 frequencies”, and “AVX-512 frequencies” [5]. Unlike the name suggests, “AVX-512 frequencies” are not triggered by arbitrary 512-bit instructions but rather only by 512-bit heavy instructions. Similarly, “AVX2 frequencies” are triggered by 512-bit light instructions and 256-bit heavy instructions. Short stretches of AVX2/AVX-512 instructions or long code sections with infrequent AVX2/AVX-512 instructions may cause less frequency reduction or may not cause any change in frequency at all [18]. After the last AVX2/AVX-512 instruction requiring the frequency reduction, the CPU waits for 670 μ s before reverting to a higher frequency [11].

3 Frequency Reduction Compensation

As described in Section 1, there are two situations in which tasks executing AVX2 and AVX-512 code can slow other tasks down [10]. First, the delay before the CPU reverts to a higher frequency means that after a context switch away from an AVX2/AVX-512 task the next task continues to execute at the lower frequency. Second, in a system with hyper-threading both hyper-threads of a physical core share the same frequency, thus a AVX2/AVX-512 task executing on one hyper-thread slows down the task executing on the other hyper-thread.

In a situation where the frequency reduction caused by power-intensive instructions is not limited to tasks executing such instructions, CPU time is not proportional to performance anymore and existing schedulers which allocate equal CPU time to tasks of equal priority fail to provide a fair distribution of CPU performance. In this paper, we take relative application performance compared to isolated execution as the main metric for fairness and consider a distribution of CPU performance “fair” if each task receives the same fraction of the CPU performance that it would receive if it was executing on the CPU alone. In a system without frequency changes and with two tasks, for example, each task would obtain 50% of the performance of the task executed in isolation. If one of the tasks executes power-intensive instructions and the other one does not, however, only the latter task (i.e., the

victim task) is affected by the resulting *remote frequency reduction overhead* [11] and receives a lower share of the CPU performance. Note that AVX2/AVX-512 tasks themselves, for example, can be assumed to obtain full CPU performance despite executing at reduced frequencies. They profit from the increased throughput of AVX2 and AVX-512 – applications have little reason to use complex power-intensive instructions if their speedup does not outweigh the frequency reduction.

Current schedulers allocate equal CPU time to individual tasks even though only some are affected by remote AVX overhead, resulting in compromised performance isolation. Some existing techniques using consumed energy as the basis for scheduling [22, 26, 31] may be applicable to solve this problem (see Section 8 for a discussion of these and other scheduling criteria). Such approaches are, however, not viable on current hardware as the CPUs lack interfaces required for sufficiently accurate energy models.

In this paper, we instead propose *frequency reduction compensation* as a simple technique to modify existing fair schedulers to reduce the impact of frequency reduction on the performance of victim tasks. To increase the performance share allocated to these tasks, we propose modifying CPU time accounting to take the performance reduction into account. If less CPU time is credited to a task, the scheduler will automatically schedule the task more often to make up for the perceived CPU time inequality. For victim tasks, we scale the CPU time credited to the tasks by the ratio between the actual average CPU frequency and the frequency at which the tasks could be executed if they were executed in isolation, so that this virtual CPU time matches the CPU throughput experienced by the tasks. The scheduler will then automatically mitigate the performance impact experienced by victim tasks to the point where all tasks receive equal shares of CPU performance. Such frequency reduction can be applied to all scheduling algorithms based on CPU time. In the following sections, we first describe how to identify victim tasks (Section 4) and then describe how to calculate the performance impact due to the frequency reduction (Section 5) before we describe important details in our implementation of the techniques (Section 6).

4 Attribution of Frequency Changes

Any scaling of the virtual CPU time must be limited to victim tasks, as only those tasks are supposed to be executed more frequently. First, we therefore need to identify whether a task is a victim task, i.e., whether it is affected by excessive frequency reduction caused by other tasks.

On recent Intel CPUs, it is trivial to determine whether the CPU frequency was reduced, as the CPUs provide performance events to count the cycles spent at AVX2 and AVX-512 frequency levels [5]. The CPUs do not provide a mechanism for the operating system to detect whether a code region triggers frequency reduction or not, though [11]. While the perfor-

mance events can be used to detect frequency level transitions, they cannot be used to identify the hyper-thread which caused a physical core to reduce its frequency. Also, during the time after a context switch the counters do not provide information about whether the current task could be executed at a higher frequency.

We therefore detect power-intensive code based on register accesses – many complex instruction set extensions introduce additional architectural registers, so accesses to these registers signal such power-intensive code. In our prototype, we use an approach found in previous work that uses a trap-based mechanism to identify AVX-512 code [10]: When the OS clears the `ZMM_Hi256` and `Hi16_ZMM` bits in the `XCR0` register, context switching for 512-bit vector registers is disabled [2, p. 13-1ff] and AVX-512 instructions trigger undefined instruction exceptions [2, p. 14-34].

Whereas previous work uses this mechanism to permanently disable AVX-512 on select CPU cores [10], we extend the mechanism to AVX2, yet we only trap the first 256-bit or 512-bit register access within a time slice to detect whether the task uses AVX2/AVX-512. At each context switch, we test whether the next task has valid 256-bit or 512-bit register content and flag the task as an AVX2/AVX-512 task if it does. If there is no valid 512-bit register state, we prevent further 512-bit register accesses as described above. If there is no valid 256-bit register state, either, we clear the `AVX` bit in the `XCR0` register as well to make future AVX2 instructions trigger exceptions. Then, if 256-bit registers are not enabled when the undefined instruction exception handler is called, we simply re-enable those registers, flag the current task as an AVX2 task and continue execution. Similarly, if 256-bit registers are already enabled but 512-bit registers are not, we enable 512-bit registers and flag the current task as an AVX-512 task. During the next scheduler invocation, we can test whether the previous task was marked as an AVX2 or AVX-512 task to determine whether to apply frequency reduction compensation.

Note that this implementation triggers two exceptions for the first AVX-512 instruction after a context switch. A more optimized implementation can reduce overhead by checking the register size of the trapped instruction and, if it accesses 512-bit registers, enabling both AVX2 and AVX-512 at once.

As Gottschlag et al. discuss, such a mechanism is not a precise indicator of whether code requires a frequency reduction [10]. As described in Section 2, the conditions for a frequency change are much more complex, so the mechanism can cause false positives as not all 256-bit and 512-bit register accesses cause a transition to AVX2/AVX-512 frequencies. Victim tasks using light AVX2/AVX-512 instructions may therefore not benefit from frequency reduction compensation even if the tasks themselves do not require reduced frequencies. In addition, AVX supports 256-bit registers as well, so our approach cannot cleanly distinguish AVX and AVX2. We discuss the impact of this limitation in Section 9.1.

4.1 CPU Feature Detection

One problem of disabling AVX2 and AVX-512 instructions via changes in the `XCR0` register is that this change breaks CPU feature detection in most applications. Intel states that before using any AVX instructions programs should first read the `XCR0` register via the `XGETBV` instruction and test whether AVX2 and AVX-512 are supported by the OS and then execute the `CPUID` instruction to test whether the required instructions are available [2, p. 14-15]. Related work suggests that this problem can be solved by virtualizing the `CPUID` instruction [10]. Recent Intel CPUs indeed provide an MSR that can be used to disable `CPUID`. However, we found it impossible to trap the `XGETBV` instruction without also disabling all vector instructions including widely used instruction set extensions such as SSE.

Instead, we propose patching all executables to replace the `XGETBV` instructions with an invalid instruction. In the invalid instruction exception handler, the kernel can then emulate `XGETBV` before returning to the application. We have verified this technique to be functional, and the additional exception causes minimal overhead given that most applications only determine the available CPU features once at startup.

In our evaluation, we wanted to compare our prototype to a stock Linux kernel, where we would not be able to execute such modified applications as the kernel lacks support for virtualization of `XGETBV`. As described in Section 7, we therefore did not integrate the technique into our prototype and instead manually modified the applications to assume that AVX2 and AVX-512 were available.

5 Calculation of the Performance Impact

Whenever a victim task has been identified, CPU time accounting for the task has to be scaled to increase the share of actual CPU time allocated to that task. As described above, the scaling has to occur in proportion to the performance impact. If we assume that the workload is CPU-bound, the performance impact p during a single scheduler time slice is defined by the ratio between the average CPU frequency experienced by the task and the average ideal CPU frequency at which the task could have been executed in isolation:

$$p = \frac{f_{\text{measured}}}{f_{\text{ideal}}} \quad (1)$$

In the case of Intel CPUs, the ideal frequency is the non-AVX frequency for non-AVX tasks and the AVX2 frequency for tasks which accessed 256-bit registers during the time slice. Note that memory-bound workloads suffer less from frequency reduction. We discuss the impact of this limitation in Section 9.2.

Whereas the average CPU frequency during a time slice is easily measurable, the ideal CPU frequency is not. Both non-AVX frequency and AVX2 frequency depend on the turbo

level which is selected by the CPU depending on the number of active cores. The number of active cores, however, can change at any point during the time slice, so counting the active cores during scheduler invocations is not sufficient to get a high-quality estimate of the average ideal frequency. Instead, we need to determine the average turbo level throughout the whole time slice. To determine the turbo level, we compare the measured frequency against each frequency expected had the chip operated at one particular turbo level during the time slice. If the measured frequency matches one of the expected frequencies, we can assume the corresponding turbo level. Else, the calculation of the ideal frequency has to be made via linear interpolation between the closest turbo levels.

As a first step, we calculate the expected frequency at a given turbo level for the measured amount of cycles spent at AVX2 and AVX-512 frequency levels. Assuming that during a time slice of length t_{total} the system spends t_i time at frequency f_i , the average frequency during the time slice can be calculated as follows:

$$f = \frac{1}{t_{\text{total}}} \sum t_i f_i \quad (2)$$

In the following, we assume that f_0 is the non-AVX frequency at the given turbo level, whereas f_1 and f_2 are the AVX2 and AVX-512 frequencies, respectively. These frequencies are published by Intel for their server CPUs [4]. As we count the cycles instead of the time spent at the three frequency levels, we substitute $t_i = c_i/f_i$ where c_i are the cycles at frequency f_i . We arrive at the following equation:

$$f(c_0, c_1, c_2) = \frac{f_0 f_1 f_2 (c_0 + c_1 + c_2)}{f_1 f_2 c_0 + f_0 f_2 c_1 + f_0 f_1 c_2} \quad (3)$$

$c_{\text{total}} = c_0 + c_1 + c_2$ is the total CPU cycle count and can be measured via a fixed-function performance counter, so only two programmable performance counters are required. If we further substitute $c_0 = c_{\text{total}} - c_1 - c_2$ as well as $r_1 = c_1/c_{\text{total}}$ and $r_2 = c_2/c_{\text{total}}$ as the ratio between the cycles spent at AVX2/AVX-512 frequencies and the total cycle count, we arrive at the final formula for the expected frequency at a specific turbo level:

$$f(r_1, r_2) = \frac{f_0 f_1 f_2}{(f_0 - f_1) f_2 r_1 + (f_0 - f_2) f_1 r_2 + f_1 f_2} \quad (4)$$

While calculating this formula requires a division, all frequencies lie within the same order of magnitude, so the formula can be calculated using fixed-point arithmetic and no floating-point arithmetic is required. The two-dimensional case of this formula for a Xeon Gold 6130 CPU under the assumption that $r_1 = 0$ – the system did not spend any time at the AVX2 frequency level – is shown in Figure 2.

At the end of each time slice, the scheduler measures the cycles spent at AVX2 and AVX-512 frequency levels as well as the average actual frequency. The formula above is then

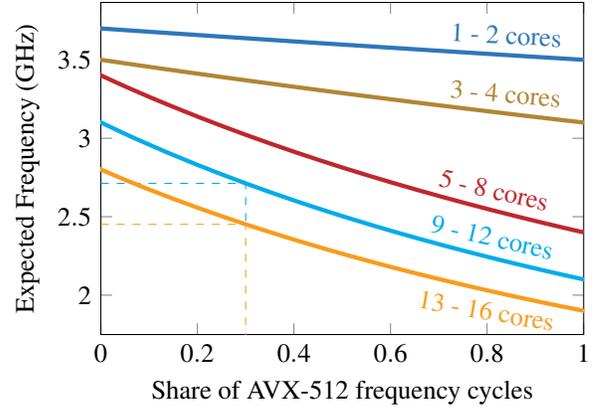


Figure 2: For a given amount of AVX2 and AVX-512 cycles during a time slice, the expected frequencies at the different turbo levels can be compared with the actual measured CPU frequency to determine the turbo level of the CPU. For example, if during a time slice 30% of all cycles were spent at the AVX-512 frequency level while an average frequency of 2.5 GHz was measured, the system likely spent most of the time at the lowest turbo level and some time at the second lowest as indicated by the dashed lines.

applied to the cycles to calculate the expected frequencies for the given number of AVX2/AVX-512 cycles at the different turbo levels. For each turbo level, the scheduler also calculates $f(0, 0)$ as the ideal frequency for non-AVX tasks and $f(r_1, 0)$ as the ideal frequency for AVX2 tasks.¹

The expected frequencies at the different turbo levels can then be compared to the measured actual frequency to determine the turbo level of the CPU, and that turbo level is used to determine the ideal frequency for the current task. If, as mentioned above, the measured frequency matches neither of the expected frequencies, linear interpolation is applied to determine the ideal frequency. The resulting ideal frequency can then be inserted into equation (1) to obtain the scaling factor for the task’s CPU time.

6 Implementation

We base the implementation of our design on the MuQSS scheduler [16], modified to use the scheduling policy of the CFS scheduler for enhanced fairness. We initially intended to use the CFS scheduler [20] as it provides particularly strict fairness for non-AVX workloads. While we show that the main scheduling policy of CFS is well-suited to our design, the implementation of CFS is not.

CFS measures the accumulated *virtual runtime* of each task, which is the actual CPU time multiplied by a priority factor [20]. The runqueue is sorted by the virtual runtime of

¹These calculations can be performed once at startup.

the tasks, and the scheduler always schedules the task with the lowest virtual runtime to ensure that all tasks are given an equal share of the CPU time. Changes to the virtual runtime as described above therefore cause preferential scheduling of victim tasks which counteracts the unfairness caused by AVX frequency reduction.

Our design, however, is incompatible with the current implementation of CFS. CFS maintains one separate runqueue per logical CPU, so any frequency reduction compensation for victim tasks only changes their priority within the runqueue of their current logical CPU. If, for example, one hyper-thread only executes AVX2/AVX-512 tasks, whereas the other only executes non-AVX tasks, the latter tasks only compete against each other during scheduling and there is no preferential treatment compared to the AVX2/AVX-512 tasks.

Frequent load balancing might improve the fairness in such a situation if an idle or more lightly loaded CPU would fetch and execute victim tasks first. In CFS, however, different cores can have different virtual runtime ranges and virtual runtimes are normalized during load balancing, so the virtual runtime advantage gained by victim tasks is often lost if tasks are migrated to a different logical CPU.

Rewriting CFS so that runtimes of tasks from different logical CPUs are comparable and introducing fast load balancing based on these runtimes is likely possible without introducing much overhead. However, we deemed the task too complex for our limited resources, so we based our implementation on the simpler MuQSS scheduler [16] which is a scheduler for Linux based on virtual deadlines. MuQSS differs from CFS in that it performs load balancing as part of the main scheduler function so that logical CPUs very frequently try to fetch tasks from other more heavily loaded CPUs.

Our tests, however, showed that MuQSS is often less fair than CFS even when no AVX2/AVX-512 is involved. We therefore replaced the deadline-based scheduling policy of MuQSS with the CPU-time-based policy of CFS, resulting in a hybrid of CFS scheduling policy and MuQSS load balancing. We then extended the policy with frequency reduction compensation as described in the previous sections. Unlike CFS, we do not perform full renormalization of the virtual runtime of tasks during load balancing. Instead, when a CPU selects a task from a different CPU, we simply limit the virtual runtime advantage the incoming task can have compared to the CPU's current lowest virtual runtime to prevent starvation of other tasks.

7 Evaluation

We evaluate our prototype to show to which degree our design can improve fairness and to determine the limitations of the design. The evaluation is conducted on a system with an Intel Xeon Gold 6130 CPU, 24 GiB of 2666 MHz DDR4 RAM, the Fedora 31 operating system, and the Linux 5.9 kernel (with the CFS scheduler or with our modified scheduler

based on MuQSS). As benchmarks for our evaluation, we use the nginx web server, most benchmarks from the Parsec 3.0 benchmark suite², as well as the Linux kernel build benchmark from the Phoronix Test Suite 9.0.1 (called “kernel-build” below). These benchmarks serve as potential victim tasks for frequency reduction compensation. In our experiments, the background application responsible for AVX frequency reduction is the x265 video encoder as it provides support for AVX-512 [29]. As described in Section 4.1, we did not implement our mechanism for CPU feature detection as part of our prototype as it would have made comparisons to an unmodified kernel much more difficult. Instead, we patch x265 to assume that AVX-512 is always available. All experiments are repeated ten times. MuQSS can be configured to share runqueues between multiple logical CPUs, and we selected runqueue sharing between hyper-thread siblings as we expected this setting to further help with the load-balancing issues described in the last section. As we show in Section 7.4, however, this setting does not seem to have much impact on the results.

7.1 Fairness

The main goal of our scheduler is to improve the fairness in a system where some tasks cause AVX frequency reduction and the performance of other tasks is affected. Fairness, in this case, means that equal CPU performance is available to the individual tasks. It is difficult to measure such fairness directly. Simply measuring the completion time of two applications when executing individually and then measuring the completion time of each application while both are executed at the same time does not yield the expected results. In particular, in a system with hyper-threading, two hyper-threads share CPU resources, and different applications may suffer differently from contention on these shared resources. An application with a large degree of instruction-level parallelism (ILP) may be able to utilize all available CPU resources when executed in isolation, but not when executed in parallel with a second application, whereas an application with little ILP may not be affected as much by other applications if the CPU resources are shared in a fair fashion. This and similar effects make it difficult to measure the unfairness caused by AVX frequency reduction.

We therefore choose a different, more indirect approach. Our experimental setup consists of two applications, a non-AVX foreground application of which we measure the completion time³, and a background application (the x265 video encoder as described above) which can be configured to use

²We excluded raytrace as it failed to finish even on a system with CFS and x264 as it showed too much variation in all experiments to provide meaningful results.

³Note that to generate HTTP requests for nginx we use the wrk2 benchmark client which has a constant benchmark duration. For the nginx benchmark, we therefore substitute the completion time with the inverse of the web server throughput (i.e., the time required per HTTP request).

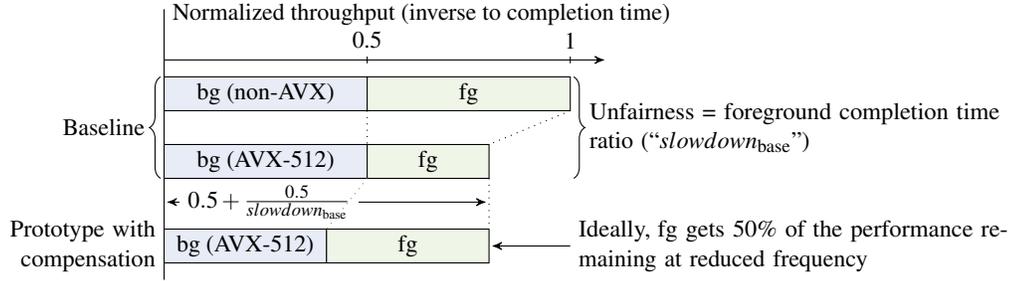


Figure 3: To calculate the fairness achieved by our prototype, we have to take into account that less overall CPU performance is available due to AVX-induced frequency reduction. Each application is supposed to be allocated half of that reduced performance.

either AVX, AVX2, or AVX-512 instructions. We start four instances of x265 with 8 threads each as otherwise x265 would not be able to utilize all available logical CPUs. Note that the completion time of x265 does not change much for the different instruction sets and that AVX, AVX2, and AVX-512 instructions are executed on the same functional units of the CPU core. Therefore, we avoid the problems described above and completion time differences for the foreground application should be representative of the AVX frequency reduction.

In a first experiment, which we call the *baseline experiment*, we execute the applications using our modified scheduler but without all code for frequency reduction compensation.⁴ We calculate the slowdown due to reduced frequencies as the completion time of the foreground application when the background application uses AVX2/AVX-512 divided by the completion time when the background application only uses AVX which does reduce the CPU frequency:

$$slowdown_{base,AVX-512} = \frac{t_{base,AVX-512}}{t_{base,AVX}} \quad (5)$$

$$slowdown_{base,AVX2} = \frac{t_{base,AVX2}}{t_{base,AVX}} \quad (6)$$

In this experiment, the slowdown provides a good metric for unfairness as no slowdown ($slowdown_{base} = 1$) means that the foreground application received the same share of CPU throughput irrespective of the choice of instructions:

$$unfairness_{base} = slowdown_{base} - 1 \quad (7)$$

We then repeat identical completion time measurements in a *prototype experiment* where we include frequency reduction compensation. As Figure 3 shows, in this case calculating the unfairness is slightly more complex. In a completely fair situation, both x265 and the foreground application would receive 50% of the CPU performance. However, overall CPU performance is reduced when x265 uses AVX2 or AVX-512, so 50%

⁴We did not compare our prototype to CFS directly as we wanted to isolate the impact of frequency reduction compensation. Related work shows that CFS does not prevent AVX2/AVX-512 tasks from slowing other tasks down [10], either, which violates the fairness definition used by this paper.

of this reduced performance are less than 50% of the CPU performance without any frequency reduction. Consequently, even with complete fairness, the foreground application still runs somewhat slower if x265 reduces the CPU frequency.

As Figure 3 shows, a slowdown during the baseline experiment of $slowdown_{base}$ results in a remaining CPU performance of $perf_{cpu} = 0.5 + 0.5/slowdown_{base}$ of the original performance. We can use this information to calculate the unfairness in the prototype experiment using the slowdown in this experiment $slowdown_{proto}$ as well as $slowdown_{base}$. If we, as in the baseline experiment, define the unfairness as the ratio between the completion time of the background application and the foreground application and then substitute the time with the inverse of the throughput, we get the following formula:

$$unfairness_{proto} = \frac{tp_{bg}}{tp_{fg}} - 1 = \frac{perf_{cpu} - tp_{fg}}{tp_{fg}} - 1 \quad (8)$$

As shown in Figure 3 the throughput of the foreground application is $tp_{fg} = 0.5/slowdown_{proto}$. Inserting into equation (8) and simplifying yields the following formula which we use in the next sections to calculate the remaining unfairness in our prototype:

$$unfairness_{proto} = slowdown_{proto} + \frac{slowdown_{proto}}{slowdown_{base}} - 2 \quad (9)$$

7.1.1 Benchmark Results

Figure 4 shows the results of the experiments described above. It can immediately be seen that our prototype greatly reduces the unfairness between the two applications. Whereas the baseline system shows an average unfairness of 7.9% for AVX2 and 24.9% for AVX-512, these numbers are reduced to 2.5% and 5.4% by our prototype, respectively.

Some benchmarks show substantially worse results than others, though. In particular, the blackscholes and dedup benchmarks show a high degree of unfairness even with our prototype. An analysis of these benchmarks shows that the benchmarks are not able to scale to all logical CPUs of the system. The blackscholes benchmark, in particular, contains long

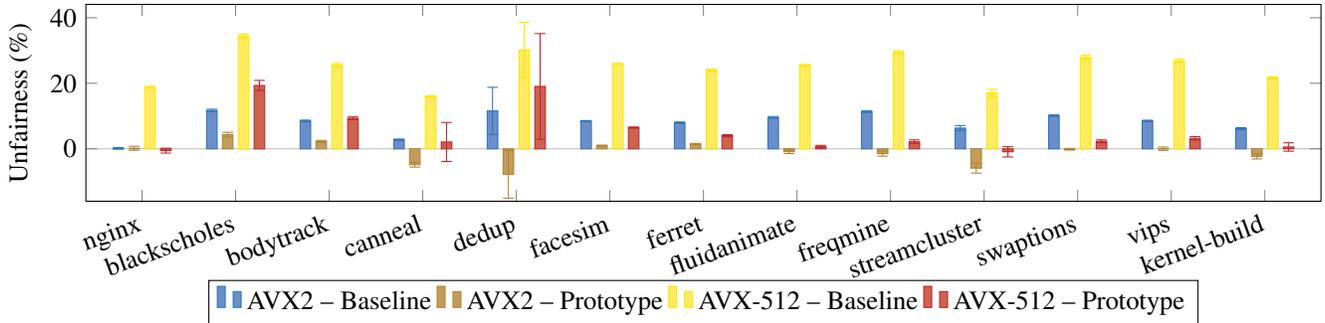


Figure 4: Unfairness for applications executed alongside x265 – for most workloads, our scheduler greatly reduces the unfairness between AVX2/AVX-512 applications and other applications executed in parallel. For AVX2, the unfairness was reduced from 7.9% (blue bars) to 2.5% (brown bars) on average, and for AVX-512 from 24.9% (yellow bars) to 5.4% (red bars).

serial phases where only one thread was active. In this case, the baseline setup already allocates almost one full logical CPU to the application. Therefore, our prototype is ineffective as it is not able to provide much more CPU time due to the lack of additional runnable threads.

This result also shows the main limitation of our approach. While overall our prototype is able to greatly increase fairness, the central mechanism – i.e., increasing the priority of victim tasks – is only effective if an increased priority translates into increased CPU time. This is not the case if an application already receives as much CPU time as it can utilize. Similarly, our approach also fails if victim application and AVX2/AVX-512 application are restricted to non-overlapping sets of logical CPUs, for example, via non-overlapping affinity masks, but effectively share physical cores via hyper-threading.

7.1.2 Synthetic Workload

Our methodology to measure the fairness for real-world applications relies on unusually complex indirect calculation of the target metric. Although the results of our experiments seem consistent, we therefore conduct an additional direct measurement of the fairness in a purely synthetic workload. For this experiment, our foreground application simply consists of 32 threads which together execute a fixed amount of 256-bit fused multiply-add (FMA) instructions requiring AVX2 frequencies, whereas the background application similarly executes a fixed amount of 512-bit FMA instructions at AVX-512 frequencies. As the two applications use very similar CPU resources, the resource contention effects described in Section 7.1 should affect both equally and should therefore not influence the observed fairness.

We try to choose the number of instructions executed by the two applications so that both require the same time if executed in isolation. We then perform a direct measurement of the fairness by comparing the completion time of the applications when executed in parallel at the same time. After one of the application finishes, we immediately start it a second time

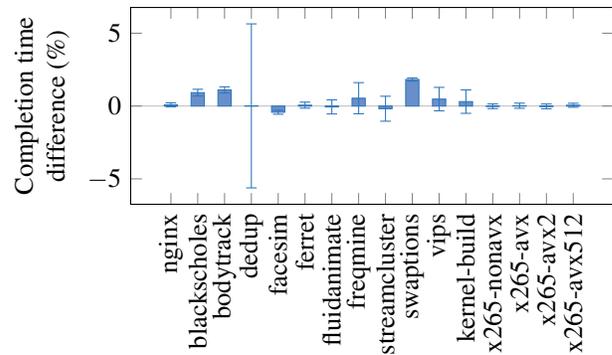


Figure 5: Our prototype causes almost no overhead when compared to a scheduler without frequency reduction compensation.

to prevent situations where only one application is running, as it then receives substantially more CPU resources. The increased throughput during that phase would mean that the measured completion time difference would not be representative for the unfairness during parallel execution.

In this setup, equal completion times are an indicator that both applications received equal shares of CPU performance. Our prototype provides almost perfect fairness. Whereas on a system without frequency reduction compensation the completion times differ by 19.2%, only a completion time difference of 0.5% remains in our prototype.

7.2 Overhead

While frequency reduction compensation can reduce the performance impact of AVX2/AVX-512 tasks on other tasks, the required modifications to the operating system can also cause overhead. In particular, the modified scheduler algorithm is executed many times per second on each core. The additional code not only increases the time spent in the scheduler itself

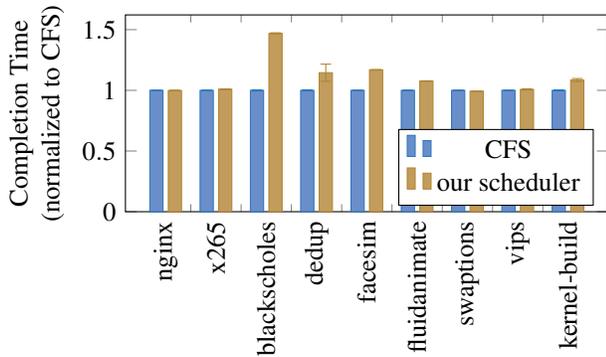


Figure 6: Our scheduler mostly provides competitive performance compared to CFS.

but also the cache footprint of the scheduler. Furthermore, the design triggers additional exceptions to detect 256-bit and 512-bit register accesses.

Most of the overhead, however, is only generated under specific circumstances. For example, the frequency impact of AVX2/AVX-512 is only calculated when the code detects a reduced frequency level at some point during the last time slice. The exceptions to detect AVX2 and AVX-512 tasks only affect such tasks and are triggered at most once per scheduler time slice.

To determine the overhead caused by our prototype, we measure the completion time of a range of benchmarks when executed in our prototype. We compare this time to the time when the benchmarks are executed with the same scheduler but without the code for frequency reduction compensation. The benchmarks include the nginx, Parsec, and PTS benchmarks used in Section 7.1 as well as the x265 video encoder as an example of an application using AVX2 or AVX-512 instructions. Due to the large variance of the dedup benchmark, we execute it 100 times to reduce the chance of misleading results.

Figure 5 shows the results of this experiment. The highest average overhead is measured for swaptions and amounted to 1.82%. All other benchmarks show less overhead, with an average overhead across all benchmarks of only 0.3%. We expect this overhead to be acceptable in most scenarios. In particular, the experiment shows that the additional exceptions to detect AVX2/AVX-512 code do not cause substantial overhead, as can be seen by the low overhead for the AVX benchmarks.

7.3 Comparison With CFS

In all experiments described above, the baseline for our measurements is our own scheduler, with the code for frequency reduction compensation removed at compile time. Even though our scheduler implements the same basic algo-

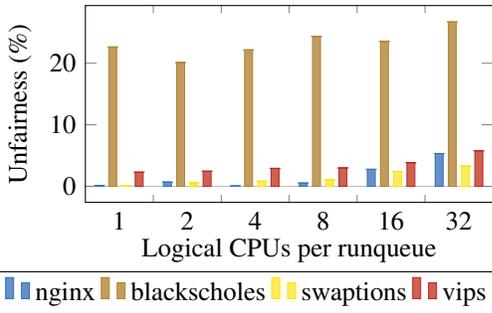
rithm as CFS, it provides a radically different implementation based on MuQSS. While MuQSS has been shown to provide similar performance as CFS [17], the implementation differences would have likely had an impact on the behavior of the benchmarks. Because we did not directly compare our MuQSS-based prototype to CFS in the experiments above, we show that the underlying scheduler – MuQSS modified to use the CFS policy, but without frequency reduction compensation – provides performance competitive to CFS. We execute a subset of the benchmarks with these two schedulers comparing the completion times.

Figure 6 shows the completion times of the benchmarks when executed with our scheduler normalized to the completion times with CFS. For most benchmarks, our scheduler causes at most 17% overhead when compared with CFS. This range matches the performance reported for unmodified MuQSS [17]. We used perf to measure the instructions per cycle (IPC) as an effort to determine the reason for the overhead. Overall, the IPC differences were much smaller than the overhead, with no clear correlation between the two, which shows that ineffective caching due to the fast load balancing required by our approach is not the main reason for the overhead. Implementing a suitable load balancing mechanism in other schedulers such as CFS is therefore unlikely to have substantial negative impact.

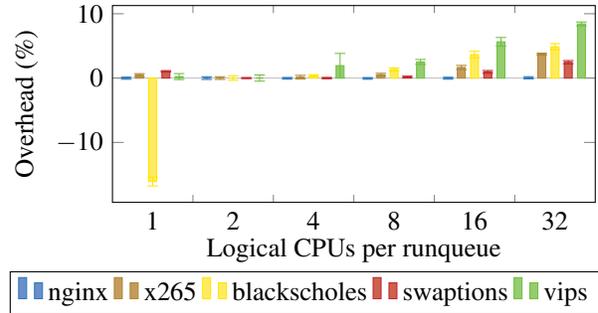
7.4 Runqueue Sharing

As described in Section 6, our implementation relies either on runqueues shared between multiple logical CPUs or on quick load balancing as otherwise the scheduler would often not have a choice between enough tasks to be able to effectively implement prioritization of victim tasks. The MuQSS scheduler used as the basis for our implementation provides both flexible runqueue sharing options as well as quick load balancing. As we want to determine whether shared runqueues were required for our approach to be effective, we measure the fairness for a subset of the benchmarks and with runqueues shared between different numbers of logical CPUs. The results of this experiment are visualized in Figure 7a, where we show the unfairness when AVX frequency compensation is enabled. Note that sharing runqueues between cores increases lock contention and potentially has a negative impact on throughput, so we also measure the overhead compared to a setup with runqueues shared between two logical CPUs. The results of this experiment are shown in Figure 7b.

Counterintuitively, sharing runqueues beyond sibling hyperthreads does not appear to have any beneficial impact on our prototype, which shows that the load balancing mechanism of our scheduler is able to provide all CPUs with enough choices so that they are able to prioritize victim tasks. The runqueue sharing options do result in slightly different throughput, however, with most benchmarks performing best if runqueues are shared between two logical CPUs. We assume blackscholes



(a) Remaining unfairness due to AVX-512 frequency reduction in our prototype (parallel execution with x265)



(b) Overhead for different runqueue sharing options compared to sharing among two logical CPUs (without frequency reduction compensation)

Figure 7: Our scheduler can share runqueues between a flexible number of CPU cores. While sharing runqueues between sibling hyper-threads – i.e., between two logical CPUs – provided the highest performance for most workloads, other configurations caused additional overhead and did not provide improved fairness.

benefits from improved cache efficiency when executed without shared runqueues. Note that despite the potential for such results the default option for MuQSS is to share one runqueue among all multicore siblings (i.e., 32 logical CPUs on our system).⁵

8 Fairness Criteria

Our evaluation shows that equal distribution of CPU time as implemented by contemporary schedulers fails to achieve its goal, which is to allocate equal CPU performance to individual tasks to improve performance isolation. We therefore propose AVX frequency compensation as a technique for fairer distribution of performance. While equal performance in terms of equal slowdown due to remote AVX overhead is an intuitive fairness definition, it is by far not the only one. We have identified two main dimensions spanning the design space of such definitions.

First, while current schedulers build upon CPU time as the main metric for fairness and we base our approach on a notion of CPU throughput, other metrics may be viable. One alternative would be to base scheduling on energy as a first-class operating system resource as suggested by related work [22, 26, 31]. Similar to our approach, such designs could penalize applications using power-intensive instructions such as AVX2 or AVX-512. While energy-based scheduling can therefore likely also solve the problem covered by this paper, it is not viable on current hardware. The CPU power model used by existing approaches either does not differentiate between executed instructions [22, 31] or uses the performance monitoring unit and requires a careful choice of performance events from which the energy can be derived with sufficient accuracy [24]. Current CPUs, however, do not provide any

⁵<https://github.com/ckolivas/linux/blob/5.9-muqss/kernel/Kconfig.MuQSS#L3>

sufficiently specific performance events for the relevant set of AVX2 and AVX-512 instructions [3, p. 19-3ff].⁶ If future hardware provides a reliable method to estimate energy usage of individual logical CPUs, energy-based scheduling may prove to be a more effective solution than our approach.

Second, the effects of CPU power management can be compensated to different degrees, ranging from no compensation at all as implemented by most current operating systems to full compensation where only power-intensive tasks are affected by frequency reduction overhead. The latter provides a degree of performance isolation that is often desired in multi-tenant systems where customers pay for specific CPU performance. However, it also actively penalizes use of accelerators or similar specialized hardware, even though such code is often more energy-efficient due to the resulting speedup [6]. Our approach implements a compromise between performance isolation and incentives to use energy-efficient accelerators: Our definition of fair distribution of CPU performance means that the frequency reduction overhead is equally shared by low-power tasks as well as the high-power tasks which caused the overhead. In the future, we envision metadata such as the CPU quota information provided by Linux cgroups to be used to choose the appropriate type of frequency reduction compensation on a task-by-task basis.

As part of such a more flexible approach, our scheduler can easily be modified to support different degrees of performance isolation. As a possible variant, we changed our scheduler to not only reduce the CPU time credited to the victim task according to the frequency reduction but to also add an identical amount to the CPU time credited to the last AVX2 or AVX-512 task executed on the physical core. Whereas the original scheduler only reduced the performance impact on non-AVX tasks by a third during the benchmarks described

⁶While the CPUs provide performance events for floating point AVX2/AVX-512 instructions, other instructions are not covered.

above, this modification foregoes fairness and allocates more CPU time to victim processes, thereby achieving an average reduction by 70%, with many benchmarks experiencing a far lower performance impact.

9 Limitations

Our evaluation demonstrates that our scheduler modifications greatly improve fairness for workloads consisting of AVX2/AVX-512 tasks and non-AVX tasks. For many of the scenarios we test, our prototype achieves 10 times better fairness than a scheduler without frequency reduction compensation. This improvement comes at near-zero cost: As we show, frequency reduction compensation has almost no overhead, and it only contributes 329 additional lines of code to our prototype. While our design therefore provides a practical solution as-is, with substantial improvements over existing schedulers, it has a number of limitations, some of which are caused by limitations of the underlying hardware. We discuss these limitations below and sketch improvements of the design where applicable.

9.1 Detection of AVX2 and AVX-512

As our design tries to increase the CPU time share of tasks which suffer from AVX frequency reduction even though they do not have to be executed at reduced frequencies, correct attribution of reduced frequencies to AVX2/AVX-512 tasks is a central part of our design. We use and extend an existing approach [10] where the CPU is configured to trigger an exception when 256-bit and 512-bit registers are accessed. This simple policy does not match the actual conditions for reduced frequencies which are substantially more complex [5, 18]. Often, usage of 256-bit and 512-bit registers is detected even though a task does not actually require the respective AVX2 and AVX-512 frequency levels. Yet, our prototype assumes that a lower frequency level is required whenever a task accesses the corresponding registers.

If, for example, a task executes light AVX-512 instructions, our prototype never applies any compensation even if the task could be executed at AVX2 frequencies as shown in Section 2. Similarly, for tasks with light AVX2 instructions which are able to execute at non-AVX frequencies, our prototype only compensates the difference between AVX2 and AVX-512 frequencies. To the best of our knowledge, there is no better hardware interface available to detect power-intensive AVX2/AVX-512 code sections. In our case, however, it is important to note that the mechanism does not cause any false negatives and therefore results in a conservative implementation where applications are never unfairly rewarded for a frequency reduction they caused themselves.

In the future, an improved software mechanism to detect AVX2/AVX-512 code may improve accuracy. For example, the operating system could scan executable pages for 256-bit

and 512-bit operations and categorize them. Then, instead of the current register-based mechanism, the OS could selectively unmap all pages containing instructions which may cause a transition to AVX2/AVX-512 frequencies to detect AVX2/AVX-512 tasks via page faults.

9.2 Memory-Bound Processes

As described in Section 5, our prototype currently assumes that performance is proportional to CPU frequency when deriving the performance impact from the calculated frequency reduction. This is an assumption that is found in related work as well [12]. The assumption is incorrect for memory-bound tasks, however. Memory latency does not increase when the CPU frequency is reduced, so memory-heavy applications suffer less from reduced frequencies [13]. Unequal performance impact due to different sensitivity to frequency changes has been identified as a challenge for fair scheduling [14].

In our case, the result is that memory-bound tasks may be given more than their fair share of the system's performance if frequency reduction compensation is applied. Consequently, the negative impact on the AVX2/AVX-512 tasks causing the frequency reduction is increased, as they in turn are allocated less than their fair share of CPU performance. This negative impact, however, is bounded: The AVX2/AVX-512 tasks always receive at least as much CPU performance as they would if all other tasks were completely CPU-bound.

In the future, our design could be extended with a more accurate performance model which takes memory accesses into account, similar to DTS [14] which has shown that better DVFS performance models can greatly increase fairness. Many such models have been described in the literature [21]. It has been shown that often very few performance counters suffice to characterize the workload and to predict performance at other CPU frequencies [25, 27]. As described in Section 5, our current technique to determine the performance impact requires at least two of the four configurable performance counters of each logical CPU (one for AVX-512 and one for AVX2 frequency level cycles) as well as a fixed-function CPU cycle counter.⁷ The two remaining configurable counters can be used to determine how memory-heavy the workload is and to implement a better performance model.

10 Related Work

To the best of our knowledge, our approach is the first fair scheduler for workloads involving AVX2 and AVX-512 tasks. However, there have been scheduling algorithms for software-controlled DVFS and there have been other techniques to mitigate the performance impact caused by AVX frequency

⁷Currently, the prototype uses three configurable performance counters. The third counter is used to determine the total number of elapsed CPU cycles and can be replaced with a fixed-function counter.

reduction. In the following section, we describe the corresponding related work and lay out the differences to our approach.

10.1 Fair Scheduling and DVFS

Existing work on fair schedulers for systems with DVFS has focused on software-controlled DVFS where the operating system chooses the CPU frequency. Two main issues affecting fairness arise from such DVFS:

1. CPU quotas are commonly expressed in terms of CPU time, and a fixed CPU time quota translates into less performance when the system is running at a reduced frequency. Hagimont et al. [12] discuss the interaction between load-based CPU frequency selection and fixed and flexible CPU time quotas. Whereas fixed CPU time quotas result in reduced fairness and insufficient CPU throughput, flexible CPU time quotas commonly prevent any frequency reduction.

Hypervisors often implement CPU time quotas in the form of credit schedulers. To solve the problems stemming from DVFS, the time per credit can be scaled by the CPU frequency and frequency selection can be modified to take flexible CPU quotas into account [12]. In this work, we use a similar approach where we implement scaling within CPU time accounting. Our work differs, though, in that we focus on hardware-controlled DVFS and on unfairness problems where one task reduces the CPU frequency experienced by another task. This scenario requires different techniques to identify the tasks affected by unfair scheduling and to determine the scale of the resulting performance impact.

In Section 8, we discussed that CPU time is not the only possible metric for scheduling. Credit schedulers in particular have been modified to scale the time per credit based on power consumption [30]. While such policies could improve fairness in workloads with AVX2 and AVX-512, the required fine-grained accurate power models cannot be constructed on current hardware for the reasons outlined above.

2. Programs suffer from the same frequency reduction to different degrees depending on how memory-heavy the programs are. Jia et al. [14] show how this affects fairness when the operating system reduces the CPU frequency. They suggest dynamic time-slice scaling (DTS) to achieve level performance, where the operating system determines the performance impact of DVFS on applications and scales time-slices accordingly. Our work solves a different problem – unfairness caused by hardware-controlled DVFS – yet uses a very similar mechanism. Instead of time-slice scaling, we modify CPU time accounting as it requires fewer modifications to the CFS

scheduling algorithm. In general, we argue, though, that the approach of DTS is orthogonal to our design. As we describe in Section 9.2, we currently assume that all tasks are affected equally by frequency reduction, so estimates of the actual performance impact as conducted by DTS can further improve the fairness of our approach.

10.2 Core Specialization

In this work, we describe a scheduler which compensates AVX frequency reduction to improve fairness. An alternative approach would be to prevent AVX frequency reduction whenever possible, as less AVX frequency reduction might also lead to improved fairness. One such technique to prevent AVX frequency reduction is to use core scheduling to limit co-scheduling of AVX-512 and non-AVX-512 tasks [19]. Similarly, core specialization [10] restricts scheduling of AVX-512 tasks to a subset of the system’s CPU cores. While, in contrast to our approach, both techniques can potentially improve overall system throughput if they achieve higher average CPU frequencies, they have the potential to cause substantial overhead. Core scheduling leaves hyper-threads idle if necessary [19], which may cause reduced utilization of CPU resources. Similarly, an earlier version of core specialization was shown to cause substantial overhead when tasks frequently had to be migrated between cores [9]. We present an approach that is far less intrusive and has less potential to cause overhead. We therefore believe that our approach is applicable to a wider range of situations.

11 Conclusion

When power-intensive instructions are executed, power-limited CPUs may have to temporarily reduce their frequency to prevent excessive power consumption. Often, this frequency reduction also affects tasks that do not execute such power-intensive instructions. Therefore, as we demonstrate, these systems require new approaches to fair scheduling.

Recent Intel CPUs with support for AVX2 and AVX-512 show such behavior, and in this work we describe a technique to identify victim tasks whose performance is affected by other AVX2/AVX-512 tasks. We describe a modification to CPU time accounting where the CPU time credited to these victim tasks is scaled according to the frequency reduction experienced by the tasks. The change causes fair schedulers based on CPU time to prioritize the tasks to the degree where they receive their fair share of CPU performance again. Our prototype is able to reduce the unfairness from 7.9% to 2.5% on average for workloads involving AVX2 applications and from 24.9% to 5.4% for AVX-512.

References

- [1] Intel® Turbo Boost Technology in Intel® Core™ Microarchitecture (Nehalem) Based Processors. Technical report, 11 2008.
- [2] *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture*, May 2018.
- [3] *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*, May 2018.
- [4] *Intel® Xeon® Processor Scalable Family – Specification Update*. Intel Corporation, February 2018.
- [5] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, September 2019.
- [6] Juan M. Cebrian, Lasse Natvig, and Magnus Jahre. Scalability analysis of avx-512 extensions. *The Journal of Supercomputing*, pages 1–16, 2019.
- [7] Edward G. Coffman, Jr. and Leonard Kleinrock. Computer scheduling methods and their countermeasures. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference (AFIPS '68 (Spring))*, pages 11–21, 1968.
- [8] Travis Downs. Gathering intel on Intel AVX-512 transitions, January 17 2020. <https://travisdowns.github.io/blog/2020/01/17/avxfreq1.html>.
- [9] Mathias Gottschlag and Frank Bellosa. Reducing AVX-induced frequency variation with core specialization. In *The 9th Workshop on Systems for Multi-core and Heterogeneous Architectures*, 2019.
- [10] Mathias Gottschlag, Peter Brantsch, and Frank Bellosa. Automatic core specialization for AVX-512 applications. In *Proceedings of the 13th ACM International Systems and Storage Conference*, pages 25–35, 2020.
- [11] Mathias Gottschlag, Tim Schmidt, and Frank Bellosa. AVX overhead profiling: How much does your fast code slow you down? In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 59–66, 2020.
- [12] Daniel Hagimont, Christine Mayap Kamga, Laurent Broto, Alain Tchana, and Noel De Palma. DVFS aware CPU credit enforcement in a virtualized system. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 123–142. Springer, 2013.
- [13] Ranjan Hebbar S R and Aleksandar Milenković. Impact of thread and frequency scaling on performance and energy efficiency: An evaluation of Core i7-8700K using SPEC CPU2017. In *2019 SoutheastCon*, pages 1–7. IEEE, 2019.
- [14] Gangyong Jia, Xuhong Gao, Xi Li, Chao Wang, and Xuehai Zhou. DTS: Using dynamic time-slice scaling to address the OS problem incurred by DVFS. In *2012 IEEE International Conference on Cluster Computing Workshops*, pages 65–72. IEEE, 2012.
- [15] Judy Kay and Piers Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, 1988.
- [16] Con Kolivas. MuQSS - the multiple queue skiplist scheduler. <http://ck.kolivas.org/patches/muqss/sched-MuQSS.txt>.
- [17] Con Kolivas. First MuQSS throughput benchmarks, October 18 2016. <http://ck-hack.blogspot.com/2016/10/first-muqss-throughput-benchmarks.html>.
- [18] Daniel Lemire. AVX-512 throttling: heavy instructions are maybe not so dangerous, August 25, 2018. <https://lemire.me/blog/2018/08/25/avx-512-throttling-heavy-instructions-are-maybe-not-so-dangerous/>.
- [19] Aubrey Li. Core scheduling: prevent fast instructions from slowing you down. Linux Plumbers Conference, September 9 2019.
- [20] Ingo Molnar. Modular scheduler core and completely fair scheduler [CFS]. *Linux-Kernel mailing list*, 2007.
- [21] Barry Rountree, David K. Lowenthal, Martin Schulz, and Bronis R. de Supinski. Practical performance prediction under dynamic voltage frequency scaling. In *2011 International Green Computing Conference and Workshops*, pages 1–8. IEEE, 2011.
- [22] Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazieres, and Nickolai Zeldovich. Energy management in mobile devices with the Cinder operating system. In *Proceedings of the sixth conference on Computer systems*, pages 139–152, 2011.
- [23] Robert Schöne, Thomas Ilsche, Mario Bielert, Andreas Gocht, and Daniel Hackenberg. Energy efficiency features of the Intel Skylake-SP processor and their impact on performance. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 399–406. IEEE, 2019.
- [24] Arsalan Shahid, Muhammad Fahad, Ravi Reddy Manumachu, and Alexey Lastovetsky. A comparative study of

techniques for energy predictive modeling using performance monitoring counters on modern multicore CPUs. *IEEE Access*, 8:143306–143332, 2020.

- [25] Vasileios Spiliopoulos, Stefanos Kaxiras, and Georgios Keramidas. Green governors: A framework for continuously adaptive dvfs. In *2011 International Green Computing Conference and Workshops*, pages 1–8. IEEE, 2011.
- [26] Jan Stoess, Christian Lang, and Frank Bellosa. Energy management for hypervisor-based virtual machines. In *2007 USENIX Annual Technical Conference (USENIX ATC'07)*, pages 1–14, 2007.
- [27] Bo Su, Joseph L Greathouse, Junli Gu, Michael Boyer, Li Shen, and Zhiying Wang. Implementing a leading loads performance predictor on commodity processors. In *2014 USENIX Annual Technical Conference (USENIX ATC'14)*, 2014.
- [28] Michael B. Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *49th ACM/EDAC/IEEE Design Automation Conference*, pages 1131–1136. IEEE, 2012.
- [29] Praveen Kumar Tiwari, Vignesh V Menon, Jayashri Murugan, Jayashree Chandrasekaran, Gopi Satykrishna Akisetty, Pradeep Ramachandran, Sravanthi Kota Venkata, Christopher A Bird, and Kevin Cone. Accelerating x265 with Intel® Advanced Vector Extensions 512. Technical report, Intel, 05 2018.
- [30] Chengjian Wen, Jun He, Jiong Zhang, and Xiang Long. PcfS: A power credit based fair scheduler under DVFS for multi-core virtualization platform. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 163–170. IEEE, 2010.
- [31] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. ECOSystem: Managing energy as a first class operating system resource. *ACM SIGOPS operating systems review*, 36(5):123–132, 2002.

SKQ: Event Scheduling for Optimizing Tail Latency in a Traditional OS Kernel

Siyao Zhao
RCS Lab, University of Waterloo

Haoyu Gu
RCS Lab, University of Waterloo

Ali José Mashtizadeh
RCS Lab, University of Waterloo

Abstract

This paper presents Schedulable Kqueue (SKQ), a new design to FreeBSD Kqueue that improves application tail latency and low-latency throughput. SKQ introduces a new scalable architecture and event scheduling. We provide multiple scheduling policies that improve cache locality and reduce workload imbalance. SKQ also enables applications to prioritize processing latency-sensitive requests over regular requests.

In the RocksDB benchmark, SKQ reduces tail latency by up to $1022\times$ and extends the low-latency throughput by $27.4\times$. SKQ also closes the gap between traditional OS kernel networking and a state-of-the-art kernel-bypass networking system by 83.7% for an imbalanced workload.

1 Introduction

Applications and hardware have evolved tremendously. Modern server applications span hundreds to thousands of nodes across the data center to serve user requests. The depth and breadth of the service tree cause users to experience request latency dominated by the tail latency of any node in an application [11]. Advancements in storage and networking are making low latency services possible [2]. Recent research systems propose using kernel-bypass and custom dataplanes to reduce latency [3, 19, 23, 36].

However, most applications in data centers are still built directly on top of traditional OSes and employ an event-driven approach, which uses an event loop that polls the kernel's event subsystem. Even popular user-level threading systems internally depend on kernel event subsystems to dispatch events efficiently across user-level threads, e.g., Go language [16] and Arachne [37].

Event subsystems in traditional OSes such as *Kqueue* [27] in FreeBSD and Mac OS X, *epoll* [29] in Linux, *IO Completion Ports* (IOCP) [9] in Windows and *Event Completion Framework* [4] in Solaris were developed nearly 20 years ago, predating many innovations in modern hardware. These event subsystems undermine features such as receive side scaling

(RSS) [8] that can improve scalability and latency. Userspace event libraries, e.g., *libevent* [32], are influenced by event subsystems and propagate these issues. Research systems such as *Megapipe* [17] and *Affinity-Accept* [34] improve event subsystems but only solve part of the problem.

The evolution of modern server applications and hardware inspired us to revisit event facilities in traditional OS kernels to address the latency problem. Improving tail latency in a traditional OS is challenging. Unlike kernel-bypass and custom dataplanes, existing kernels are multipurpose and complex. Kernel event facilities tightly integrate with many subsystems including storage, network, and process management. Furthermore, we must carefully tradeoff our system's overhead with the performance benefit to the rest of the system.

This paper presents Schedulable Kqueue (SKQ), a novel event notification facility based on FreeBSD Kqueue that provides a more flexible abstraction, and allows applications to use features of modern hardware. SKQ improves tail latency and extends low-latency throughput. SKQ introduces a new architecture with improved scalability, fine-grained event scheduling and event delivery control. Application threads share a single, scalable SKQ instance, which automatically schedules events across all threads. While improvements to kernel event notification facilities have been proposed [17, 34], SKQ differentiates itself with the following contributions:

- SKQ offers multiple application-controlled scheduling policies to improve cache locality and workload imbalance. We present guidelines for policy selection in § 4.3.
- SKQ presents explicit control over event delivery including event pinning and prioritization.
- SKQ has been extensively tested and deployed on production servers. We also developed event libraries to facilitate integration in existing applications.

We evaluate SKQ with microbenchmarks and multiple real world workloads with different characteristics. We examine workloads with uniform and non-uniform request service times, and IO-bound workloads. In microbenchmarks, we

show that SKQ reduces lock contention, improves cache locality and multicore scalability. In our RocksDB [14] benchmark running the Facebook ZippyDB workload, SKQ extends the low-latency throughput by 27.4× and reduces latency by up to 1022×. Event prioritization allows a saturated server to service low-latency requests to a high priority client with 8× lower latency while having little performance impact on other clients. Additionally, SKQ closes the gap between traditional OS kernel networking and a state-of-the-art kernel-bypass networking system by 83.7% for an imbalanced workload.

2 Background and Related Work

SKQ was motivated by the design and performance problems with existing event subsystems. While our observations hold on popular platforms, i.e., Linux `epoll` and Windows IOCP, we will explain everything in the context of FreeBSD Kqueue.

Sources of Latency: Li et al. [28] studied several server applications and identified multiple factors in the OS that contribute to high tail latency. The authors suggested reducing background tasks, pinning threads to cores and avoiding NUMA effects. Even after following all the recommendations, we found two additional factors that lead to high tail latency.

One factor is cache misses due to receive side scaling (RSS) [8] found in modern NICs. RSS creates a send and a receive NIC queue per core and distributes connections across them using hash functions. Recent implementations of RSS also load balance by migrating groups of connections between cores. However, applications are unable to detect this and will process the connection on the original core, losing connection affinity and causing cache misses. In our Memcached benchmark, we found that up to 77% of total L2 cache misses are due to improper connection affinity and are avoidable. SKQ maintains connection affinity and follows connection migration with the *CPU affinity* scheduling policy.

Another factor is workload imbalance that arises from differences in request service time and suboptimal connection distribution across worker threads. Workload imbalance over-saturates some worker threads while under-saturating others. To put this into perspective, we measured the difference in total processing time between the most and least busy threads in two workloads. In Memcached, a uniform workload, we measured a 2.8% difference. In a GIS application with a Zipf-like service time distribution, we measured a 46% difference. As a result, the GIS application's tail latency increases much faster than Memcached as a function of throughput. SKQ offers scheduling policies to minimize workload imbalance.

Event-Driven Models: Most modern event-driven applications use either the *1:1 model* or the *1:N model*.

Applications using the 1:1 model create one Kqueue per thread where connections are assigned to thread private Kqueues. This model scales well and maintains connection affinity. However, the 1:1 model suffers from two problems.

First, it hinders efficient event migration. Moving events between Kqueues requires two system calls that remove events from the source Kqueue and add them to the target Kqueue. This migration process also involves multiple kernel and userspace locks, leading to poor scalability. Second, this model interacts poorly with RSS in modern NICs as applications cannot detect nor react to RSS connection migration.

The 1:N model means all threads share a single Kqueue and process connections in a round-robin fashion. This model maximizes CPU utilization and simplifies event scheduling, but suffers from two problems. First, the model leads to lock contention on multicore machines (see § 6). Second, the model does not preserve connection affinity as each connection is processed by different threads, which results in cache misses. As a result, applications and event libraries have mostly avoided the 1:N model except for a few low-throughput services [25].

SKQ uses a hybrid architecture that offers the best of both worlds through our event scheduler. SKQ exposes a 1:N model to applications for efficient event scheduling and delivery while internally using a 1:1 model for multicore scalability.

Need for Application Control: Applications often need to deliver events to a specific thread. For example, one thread needs to notify another thread of control events. Kqueue does not provide applications using the 1:N model with fine-grained event delivery control. SKQ allows applications to pin events to specific threads.

SKQ also introduces application controlled event prioritization, which allows applications to prioritize latency-sensitive events over batch-processing events. To our knowledge, no existing kernel event subsystem supports event prioritization. The latest generation NICs provide hardware support for event prioritization with application device queues (ADQs) [20]. SKQ can be used with ADQs to improve event prioritization throughout the networking stack.

RFS, RPS and Intel Flow Director: Receive Flow Steering (RFS) [12] and Receive Packet Steering (RPS) [6] are Linux networking stack features. RFS improves connection locality by enabling the kernel to process packets on the core where the corresponding application thread is running. RPS is a software implementation of hardware RSS, which provides hash-based packet distribution across cores. Intel Flow Director [18] is a hardware implementation of RFS. SKQ eliminates the need for RFS with the *CPU affinity* policy.

io_uring: `io_uring` [7] is a recent Linux kernel feature that enables efficient asynchronous IO and avoids system call overhead via polling. Each `io_uring` object uses a single submission queue and a single completion queue for kernel-user communications. However, `io_uring` does not provide event scheduling between threads, which is the goal of SKQ.

FreeBSD offers the `lio_listio` [41] interface for asynchronous IO, which batches rather than eliminates system calls. We benchmarked the benefit of `lio_listio` combined with SKQ to improve low-latency throughput by roughly 40%,

compared to 19.8% with SKQ alone on our web server benchmark (§ 6.7). Our techniques are also applicable to `io_uring`.

Windows IOCP: Windows IOCP is functionally similar to Kqueue and provides a developer friendly API for blocking IOs when used with the 1:N model [10, 38]. Each IOCP internally uses one event queue and provides a concurrency parameter to limit the number of running threads. Threads beyond this limit block until a running thread sleeps on an unrelated IO operation (e.g. disk IO) [9]. IOCP also suffers from lock contention at high core counts because of the 1:N model. As a result, only applications with long running requests, such as Exchange and MSSQL [25], use the 1:N model.

MegaPipe: MegaPipe [17] permanently affinizes accepted connections to threads to improve cache affinity. MegaPipe internally uses the 1:1 model. Prior to MegaPipe, Affinity-Accept [34] and Chronos [26] also suggest maintaining connection affinity. MegaPipe delivers socket data along with triggered connections to reduce system call overhead.

SKQ's *queue affinity* scheduling policy is similar to MegaPipe's behavior. However, queue affinity is inferior to CPU affinity that takes into account connection migration in kernel and NICs. MegaPipe was developed much earlier when connection migration was less frequent due to the maturity of RSS implementations.

Custom Networking Stack: Arrakis [35], Chronos [26], DPDK [19], F-Stack [40], IX [3], and mTCP [22] use kernel-bypass to reduce system calls and/or kernel networking stack processing overhead. Other systems such as Snap [30], Shinjuku [23], and ZygOS [36] apply thread scheduling on custom dataplanes. Shenango [33] implements user-level scheduling on top of DPDK and uses work stealing for load balancing.

SKQ differs from the systems above in three ways. First, SKQ schedules events between threads rather than threads between cores. Second, SKQ offers control over event delivery and scheduling policies in addition to work stealing. Last, SKQ is implemented in a traditional OS where most server applications are still being developed. SKQ enables applications to gain performance benefit with minimal changes. The systems above however generally suffer from adoption issues.

3 Design Overview

Modern applications require increasingly better multicore scalability, cache locality and scheduling support. FreeBSD Kqueue was designed to solve the C10K [24] problem. Server applications are now running with many cores and millions, rather than thousands, of clients. The changes in applications' scale and requirements require us to revisit Kqueue's design.

The current design presents three major issues. First, using Kqueue in the 1:N model results in scalability bottlenecks due to lock contention in some multithreaded applications with as few as 4 threads. Second, we observed cache misses and load imbalances that could not be solved. Finally, Kqueue processes events in FIFO order and lacks event prioritization.

SKQ is designed with three primary goals: scalability, event scheduling, and event prioritization. SKQ is also designed to be compatible with the Kqueue API.

- **Scalability:** SKQ scales to multicore machines and a large number of events. Our design allows a single SKQ to efficiently schedule events to multiple threads with little overhead and lock contention.
- **Event Scheduling:** SKQ implements scheduling policies that improve cache locality and minimize workload imbalance. Applications can select policies based on workload characteristics to minimize tail latency.
- **Event Prioritization:** SKQ enables applications to prioritize processing high priority events versus regular events, with minimal performance impact.

One major challenge is providing low overhead scheduling as any overhead can impact application performance. To put this in perspective, for Memcached, an increase in processing times of 150 cycles would result in a 1% drop in peak throughput, while a single L3 cache miss reduces throughput by 2.7%. Scheduling introduces unavoidable overhead due to statistics collection and making scheduling decisions.

3.1 Kqueue

The Kqueue API consists of two system calls. The `kqueue()` system call creates a Kqueue kernel object and returns a file descriptor. The `kevent()` system call handles event registration, update and delivery.

Applications manipulate events by calling `kevent()` on a Kqueue object with the file descriptor of a kernel object and the type of event to monitor (e.g. read availability on a socket). For event registration, the target Kqueue object allocates a *knote* to track the information and the state of the registered event. For fast lookups, all knotes on a Kqueue are kept in a hash table in the Kqueue object.

Upon creation, the knote is also attached to the kernel object of interest. In FreeBSD, all kernel objects supported by Kqueue maintain a list of attached knotes. When the kernel object triggers an event, it activates all attached knotes by notifying their corresponding Kqueue objects.

Kqueue enqueues activated knotes into a single event queue protected by a giant lock. Application threads retrieve events with the `kevent()` system call and a userspace buffer used to hold the returned events. Internally, all threads acquire the giant lock and dequeue knotes from the event queue.

One correctness guarantee that a shared Kqueue provides is to avoid returning an event that is being processed by a thread to another thread. For example, a shared Kqueue must not notify more than one thread of read availability on the same socket, otherwise a race may occur. Kqueue achieves this guarantee by relying on application threads marking shared events as *dispatch events*, which instruct Kqueue to disable the event after returning it to the application. Once an application

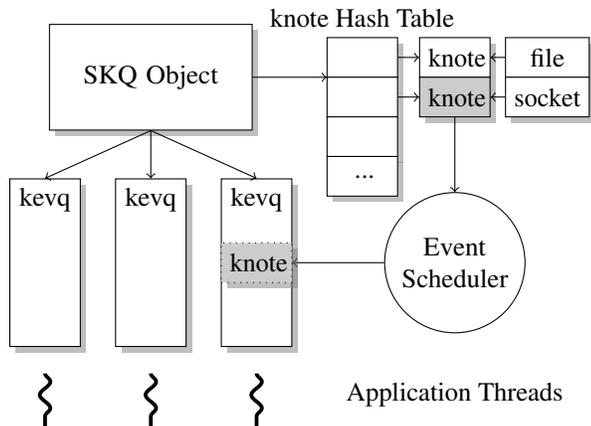


Figure 1: System Overview. SKQ creates thread private kevm to reduce lock contention and introduces an event scheduler. In this diagram a socket activated a knote and the event scheduler enqueued it into a specific kevm based on an application defined scheduling policy.

thread finishes processing a shared event, the thread must call `kevent()` to re-enable the event so it can trigger again.

Both the single event queue design and frequent event enabling/disabling cause lock contention. Therefore, a Kqueue is rarely shared between threads in an application. Additionally, using a single event queue hinders optimizing for cache locality [17,34]. User-level scheduling is difficult as migrating knotes between Kqueues is cumbersome and inefficient.

3.2 SKQ

SKQ provides applications with event scheduling and delivery controls by sharing a single scalable instance between all application threads. Figure 1 shows the overall architecture. SKQ extends Kqueue in three main ways: employing a new scalable design, offering event scheduling and delivery control, and optimizing the event lifecycle. Additionally, SKQ maintains a compatibility mode that exhibits the same behavior as Kqueue.

Scalability: SKQ improves multicore scalability and enables efficient event scheduling by introducing a new internal, lightweight, per-thread *kevm* structure. Each SKQ creates one kevm for each application thread, which is an event queue that holds knotes assigned to the corresponding thread.

When applications query events, each worker thread locks and retrieves knotes from its private kevm, eliminating a major source of contention on the giant Kqueue lock. Furthermore, kevm allow SKQ to quickly schedule events between lightweight internal structures instead of kernel objects. Implementing event scheduling between kernel objects requires extra locking (e.g. file descriptor locks) and complicates resource cleanup. SKQ approximates the benefit of the 1:N model without scalability bottlenecks or poor cache affinity

through scheduling on top of an internal 1:1 model.

When an SKQ is destroyed, all of its kevm are destroyed. If a thread exits while an SKQ is still active, only the exiting thread's kevm is drained and destroyed. Knotes already queued to the kevm are rescheduled to other available threads.

Applications are allowed to create multiple SKQs at a time, meaning each application thread can correspond to multiple kevm belonging to different SKQs. We use a hash table that maintains the mapping from kevm to their respective SKQs in the kernel thread object.

The per-thread design allows SKQ to more easily handle applications that spawn greater or fewer threads than available cores, and thread migration. While it might seem natural to use a per-core design, some applications process blocking IO calls and have more threads than cores. Applications with fewer threads than cores would require extra handling to prevent events from being left on cores with no threads.

Event Scheduling and Delivery Control: SKQ introduces an event scheduler to schedule knotes both passively and actively. Upon event activation, the event scheduler determines the target kevm of the activated knote based on the selected scheduling policy. When a kevm has no active knotes, the kevm also actively tries to steal knotes from other kevm. Applications can control the scheduling policy via `ioctl()` and new per-event flags for pinning and priorities.

SKQ also allows applications to mark individual events as high priority. SKQ favors returning high priority events first. Two tunables are provided to control the exact behavior. Applications can adjust the tunables based on their requirements. We elaborate on both features in § 4.

Optimized Event Lifecycle: We optimize SKQ's event handling to eliminate the need of re-enabling events after processing and to improve scheduling fairness.

SKQ adds a processing flag to knotes and a processing queue to kevm. Each processing queue holds knotes returned to the application from the last `kevent()` call. SKQ marks these knotes as processing, which can be scheduled to a different kevm although they are temporarily ignored by event queries. When the original thread finishes processing events, it calls `kevent()` again, which releases all knotes on its processing queue so they can be returned by future event queries.

Applications do not need to re-enable events because SKQ marks them as processing in the kernel and will not deliver them to other threads until the original thread releases them. This optimization improves scheduling fairness as processing events are always scheduled on activation rather than after the original thread returns, preserving the relative order of arrival.

4 Event Scheduling Policies

The event scheduler selects the kevm for an activated knote based on the scheduling policy. SKQ offers two categories of scheduling policies that improve cache locality and reduce

workload imbalance, which can be combined. Additionally, applications can pin events to threads and prioritize events.

We originally planned to provide a single policy that would perform well for all workloads. However, we realized from our experiments that the best scheduling policy is application and workload dependent (§ 4.3 describes policy selection). To this end, SKQ provides applications with control over scheduling policy to better meet the developer and user needs.

The biggest challenge is balancing the overhead and the optimality of making scheduling decisions. Since the event scheduler runs on every event activation, we must minimize the overhead so that the cost of scheduling does not overshadow its benefit. Towards this goal, we carefully designed SKQ while considering the impact of lock contention, CPU overhead and cache footprint.

4.1 Cache Locality Policies

Cache misses are detrimental to applications. For example, a read request in Memcached takes about 15k cycles from the NIC receiving the request to sending the response on our machine, while a L3 cache miss takes 400 cycles. This means that each cache miss increases processing time by 2.67% and correspondingly reduces throughput. SKQ provides two policies to help applications improve cache locality.

Both policies use our *kqdom* structure, which is a multi-level N-ary tree that mirrors the system's cache and memory topology, and keeps track of the core affinity of all kevs. Each *kqdom* leaf node corresponds to a core and contains a list of local kevs. Each *kqdom* level represents a shared cache level. During initialization, each SKQ creates a private *kqdom*. When an SKQ object registers a thread, the kevs is inserted into the corresponding *kqdom* leaf node. When the CPU scheduler reschedules a thread to a different core, the kevs is also moved to the new *kqdom* leaf node.

On a multi-socket machine, a *kqdom* typically contains 3 levels. The top level nodes represent different NUMA domains. The second level contains all cores that share the last-level cache within a NUMA domain. The third level consists of leaf nodes corresponding to a core containing hyperthreads.

Queue Affinity: The *queue affinity* policy always delivers events to the core where the event first triggered, which reduces userspace cache misses by maintaining affinity.

When a knote is activated for the first time, SKQ stores the corresponding *kqdom* leaf node in the knote. On subsequent activations, the event scheduler queues the knote to a kevs in the same *kqdom* leaf node. When multiple kevs are present in the same *kqdom* leaf node, a random kevs is selected. However, when connections migrate, this policy results in more cache misses in the kernel.

CPU Affinity: The *CPU affinity* policy delivers events to the application threads local to the triggering core. For network sockets, the policy always queues knotes to the core that received the NIC interrupt and processed the packet.

This policy improves cache locality within the kernel. Before an event activation, the networking stack has already processed the incoming packet, which pulls socket buffers and metadata into the local core's cache. CPU affinity policy keeps these cache lines local so subsequent reads and writes in userspace will more likely result in cache hits.

Furthermore, CPU affinity cooperates well with all sources of connection migration including RSS. After a connection migrates to a different core, the policy follows the migration and queues the event to a kevs on the new core. This reduces the cache misses caused by the application and kernel processing the event on different cores. This also reduces kevs lock contention during event activation as cores will deliver events to their local kevs from the packet processing, rather than competing to deliver events to the same kevs.

4.2 Workload Balancing Policies

Workload imbalance is another major cause of suboptimal performance. An imbalanced load distribution leads to resource under-utilization where some worker threads are idling whereas others are overloaded. We implement two policies that mitigate imbalances.

Best of Two: The *best of two* policy load balances events between threads by randomly selecting two kevs and choosing the kevs with a lower expected wait time. Mitzenmacher [31] shows that this policy offers good optimality. Expected wait time $E[t]$ refers to the wait time before the activated knote is serviced by an application thread.

For each kevs, we maintain the number of knotes currently queued n_{cur} , the number of knotes returned to userspace of the last *kevent()* call n_{ret} , the timestamp of the last *kevent()* call t_{ret} , and a moving average of the processing time in cycles per knote t_{avg} . We use an exponential moving average with $\alpha = 0.05$, hence $t'_{avg} = 0.05 * t_{new} + 0.95 * t_{avg}$, which we found to react fast enough while smoothing out workload behavior and noise from interrupts and scheduling. When applications finish processing a knote and return it to SKQ via *kevent()*, the moving average is updated.

On knote activation, the event scheduler calculates the expected wait time with $E[t] = t_{ret} + t_{avg} * (n_{ret} + n_{cur})$ for both selected kevs. Finally, the event scheduler enqueues the knote to the kevs with earlier $E[t]$.

A potential issue is that a thread may unexpectedly stall in userspace. In this case, $E[t]$ can lag far behind and be in the past. To prevent the event scheduler from assigning too many knotes to the thread, we set t_{ret} to $MAX(t_{ret}, t_{cur} - t_{avg} * n_{ret})$, where t_{cur} is the current timestamp.

Work Stealing: Work stealing allows idle threads to steal knotes from another thread's kevs. Unlike other scheduling policies, work stealing operates during the dequeuing of knotes rather than knote activation. When a thread has no knotes to process, it normally blocks until some knotes arrive. With work stealing enabled, instead of blocking, the idle

thread picks a random victim ke_{vq} and tries to steal knotes. The system will periodically retry until either new knotes arrive or at least one knote is stolen. Applications can control the maximum stolen knotes per attempt (defaults to 1).

In order to reduce the overhead and lock contention of work stealing, we use trylocks to check the availability of the victim ke_{vq} and knotes. The idle thread first trylocks the victim ke_{vq}. If the victim ke_{vq} is busy, the idle thread sleeps for a fixed amount of time then retries. Otherwise, the idle thread scans the victim ke_{vq} for knotes. We use the same trylock technique to probe each knote, skipping knotes that are undergoing changes (e.g., being scheduled). To avoid locking the victim ke_{vq} for too long, we bound the maximum number of scanned knotes to twice the maximum stolen knotes.

Another issue is thrashing that can occur because of lock ordering issues. When a knote is stolen by an idle thread it is moved to the idle thread's ke_{vq}. For a short window of time we must drop all ke_{vq} locks to not violate lock ordering. This allows another thread to steal events from the idle thread before it can process any stolen knotes. This causes thrashing of events as they can bounce around due to this race. We added a flag to denote whether it is stolen. Knotes with the flag set are skipped during work stealing. The flag is cleared after the stolen knote is processed by the idle thread.

Our measurements show that for applicable workloads, work stealing improves the latency response and reduces tail latency (see § 6.5). We also observe negligible lock contention because of the trylock optimization using FreeBSD *lockstat*. Our earlier implementation did not employ the trylock optimization that caused work stealing to increase the overall latency in some workloads due to the aforementioned issues.

Hybrid Policies: SKQ allows applications to combine cache locality policies with workload balancing policies. In this case, the event scheduler picks the first ke_{vq} according to the cache locality policy, and then uses best of two to pick a second ke_{vq} from the rest. The expected wait time of both selected ke_{vqs} are then compared with a constant cache miss penalty applied to the ke_{vq} selected by best of two. While the cache miss penalty of migrating to a random core is both application and workload dependent, in our experiments we found a constant penalty was enough to prevent unnecessary migration when the imbalance was not significant.

Work stealing can be used with other policy combinations as it operates during knote dequeuing. In our experiments, hybrid policies that combine all three scheduling options comprise the best-performing policies for imbalanced workloads.

4.3 Policy Selection Criteria

Policy selection is based on the application's workload characteristics. Applications with uniform and low response times should use the CPU affinity policy to maximize cache affinity. Applications with imbalanced or IO-heavy workloads should use the hybrid policy consisting of CPU affinity and best of

two with work stealing to balance the threads and extend low-latency throughput. The rationale is discussed in § 6.

4.4 Fine-grained Event Delivery Controls

Besides scheduling, SKQ allows applications to control event delivery on a per-event level. We currently offer two controls to handle event pinning and event prioritization.

Event Pinning: SKQ allows applications to pin individual events to specific threads. Application threads use the affinity flag for each event during event registration. The flag ensures that the event is always delivered to the registering thread.

This is useful to applications where threads communicate based on pipes or user events. For example, in Memcached, the main thread uses a pipe to communicate control messages to each worker thread. Scheduling these events between threads would break the notification system.

Event Prioritization: Event prioritization enables applications to prioritize latency-sensitive traffic such as end user requests over batch processing. SKQ defines two event priority levels: regular and high priority. By default, all registered events are of regular priority. Applications can promote an event to high priority by setting the high priority flag.

In each ke_{vq}, we maintain a separate event queue for high priority knotes. During event activation, the event scheduler queues activated knotes to their corresponding queue. During event queries, `kevent()` prioritizes dequeuing knotes from the high priority queue. The remaining space left in the userspace buffer is filled with regular priority events.

Our design also addresses two potential problems of event prioritization. First, too many high priority events may cause starvation. To prevent starvation, we introduced the *rtshare* parameter that controls the maximum percentage of returned high priority events. For example, an *rtshare* of 80 means at most 8 high priority events out of 10 total events will be returned to the application per `kevent()` call.

Second, if `kevent()` returns too many events per call, which is common at high throughput, the application thread might spend a long time processing all events, postponing the delivery of newly arrived high priority events. To address this issue, applications can set the *rtfreq* parameter to control the number of `kevent()` calls a thread should perform per second, bounding the latency of high priority events. SKQ dynamically limits the number of events returned to the application based on the average processing time statistics.

Applications gain the full benefit of event prioritization by tuning *rtshare* and *rtfreq* based on workload characteristics and user requirements. In our benchmark, we observed an 8× tail latency improvement of a high priority client at peak throughput with little impact on regular clients.

5 Implementation

We implemented SKQ in FreeBSD 13 (commit 04816a1) with ~3000 source lines of code (SLOC). We also developed two SKQ event libraries *libevent-skq* and *libsq* for easy adoption in ~1700 SLOC. *libevent-skq* is compatible with *libevent* and solves its limitation with concurrent event processing on a single event base. *libsq* uses a custom API with reduced lock contention and system call batching via *lio_listio* (disabled in our evaluation). Additionally, we implemented a custom webserver in ~1000 SLOC on top of *libsq* that conforms with HTTP/1.1 as defined in RFCs 7230–7237 [21].

6 Evaluation

We evaluate our system using microbenchmarks and four applications. We choose workloads based on their request service time characteristics, e.g. Memcached has a uniform request service time whereas RocksDB displays workload imbalance. We also measure the event prioritization benefit and compare SKQ with Shenango, a kernel-bypass system.

All server applications run on a server with dual 2.1 GHz Intel Skylake-SP Silver 4116 with an Intel X722 10 GbE NIC. The workload generating clients run on 6 identical Skylake machines and 4 additional machines with dual Intel Xeon E5-2680 and a Mellanox ConnectX-3 10 GbE. Turbo boost is disabled on all machines to minimize measurement error. Hyperthreading is enabled with one application thread per core and NIC interrupts scheduled to the adjacent hyperthread as recommended by NIC vendors.

In all experiments we use 12 server threads, 12 client threads and 12 connections per client thread. Each run consists of a 5-second warmup followed by a 25-second measurement. All threads are pinned to the first socket. The measurement client establishes one connection per thread to probe the server response time. Each data point is obtained as the average of three runs and all results are statistically significant.

In the following sections, *vanilla* refers to multiple SKQs (the 1:1 model) to isolate the scheduling benefit. We do this because SKQ in compatibility mode provides a modest performance improvement over Kqueue (§ 6.2).

6.1 Scalability

We built a benchmark that generates events via UNIX pipes to measure the scalability of Kqueue/SKQ. Each client thread establishes 12 pipe pairs and sends requests with a constant processing time of 5 μ s. We run four configurations in total while varying the number of threads.

Shared SKQ scales linearly and on par with both multiple Kqueues and multiple SKQs. We see a constant throughput increase with each additional core, since all three setups have little lock contention and SKQ schedules events with low overhead. Shared Kqueue starts to contend after 4 cores. At

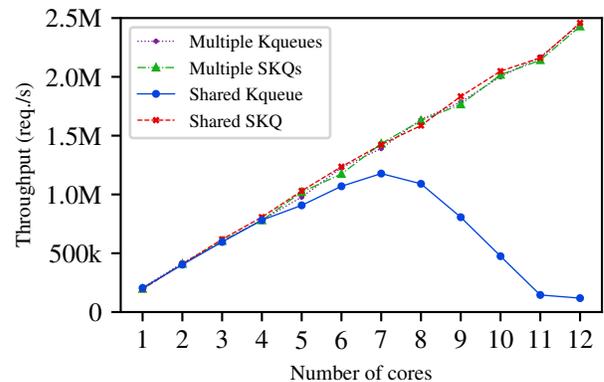


Figure 2: The impact of adding cores on various configurations’ peak throughput. Shared refers to the 1:N model while Multiple refers to the 1:1 model. We use the CPU affinity policy for shared SKQ.

12 cores, shared Kqueue only achieves 4.8% of the throughput of other configurations. This reduction in throughput comes from lock contention in Kqueue. Additional experiments show that the result of shared Kqueue is workload dependent. For requests with very short processing time, the giant Kqueue lock is the main source of contention. The lock contention decreases as the processing time increases.

We use FreeBSD *lockstat* [15] to compare the lock contention of shared Kqueue with SKQ at maximum throughput with 12 cores. The result shows that shared Kqueue has significantly higher lock contention due to threads competing for the single event queue and the giant lock.

For shared Kqueue, the top three contending system locks belong to Kqueue, which comprise 69% of total lock contention. Shared SKQ displays much lower contention with the worst contending SKQ lock ranked the 4th in the system. The top three contending SKQ locks only comprise 12% of total lock contention.

By breaking up Kqueue’s giant lock and introducing thread private *kevqs*, SKQ significantly lowers the lock contention at high core counts, achieving much better scalability.

CPU Affinity vs. Queue Affinity: In our benchmarks, CPU affinity performs on par or better than queue affinity. As we mentioned before, this is due to connection migration, which breaks connection affinity with queue affinity. *Thus, we consider CPU affinity superior to queue affinity in general and will omit queue affinity from further discussions.*

6.2 Multiple SKQs vs. Multiple Kqueues

Figure 3 shows the latency response of an unmodified Memcached using multiple Kqueues vs. multiple SKQs (the 1:1 model). We improve tail latency at throughputs between 750k–1.1M req./s. At maximum throughput, we lower tail latency by 33%. The improvement comes from the new SKQ archi-

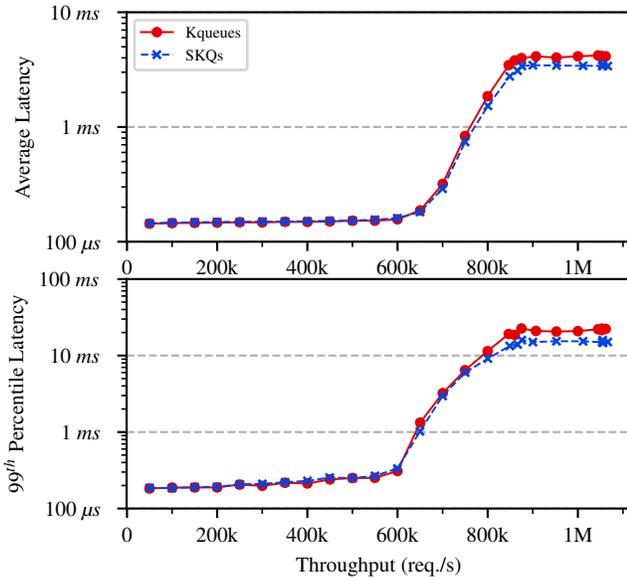


Figure 3: The latency response of Memcached using multiple SKQs vs. multiple Kqueues. SKQ lowers the tail latency by 33% at high throughput due to fine-grained locking.

ecture and fine-grained locking.

Throughout the evaluation, "vanilla" refers to multiple SKQs (the 1:1 model) to accurately measure the benefit of our scheduling policies. This isolates the improvement due to architectural changes.

6.3 Cache Miss Analysis

We use an RPC echo service to understand how cache locality policies affect cache miss rates. Figure 4 shows the latency response for an RPC echo client while varying throughput against three different policies. Both CPU and queue affinity outperform vanilla above 500k req./s. At 580k req./s, both policies show the maximum benefit over the vanilla with a $6.8\times$ ($269\ \mu\text{s}$ vs. $1839\ \mu\text{s}$) lower tail latency and 20% more low-latency throughput (585k req./s vs. 487k req./s).

Using CPU performance counters, we analyze the cache behavior to understand where the benefit comes from. Table 1 shows the L2 cache misses at 580k req./s broken down by various kernel code paths.

Both scheduling policies significantly reduce L2 cache misses in the TCP input/output path and event query path. The TCP input path includes receive-side packet processing, while the TCP output path includes processing the socket buffer into packets. The event query path is all the code executed during `kevent()` calls to retrieve events. In all paths, SKQ outperforms as both cache locality policies preserve cache affinity better, thus reducing cache misses.

The vanilla configuration of Memcached (see § 6.4) is typical of other applications that ignore cache affinity information

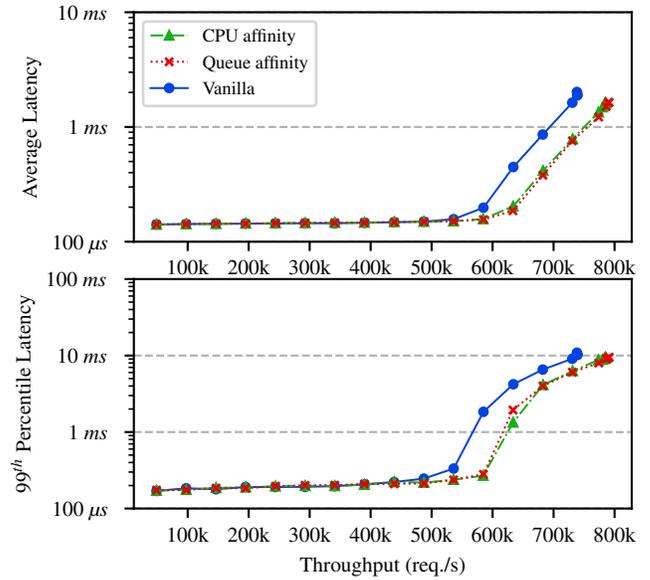


Figure 4: Cache locality policies' latency improvements of a lightweight RPC echo endpoint.

Policy	TCP input	TCP output	Event activation	Event query	Total
CPU	252k	15k	63k	166k	496k
Queue	343k	33k	95k	250k	721k
Vanilla	828k	76k	45k	1235k	2184k

Table 1: L2 data cache misses collected using hardware performance counters for 20 s running at 580k req./s. TCP input/output refer to the packet transmit and receive path, event activation includes event activation and scheduling, and event query is the application retrieving events with `kevent()`.

(unavailable in userspace) and use a round-robin assignment of connections to worker threads. As a result, vanilla contains many mismatched connections, i.e. connections processed by the kernel on one core but processed by an application thread on a different core, leading to more cache misses and worse tail latency.

In the event activation path, vanilla has fewer cache misses than our scheduling policies. This is expected as both policies require extra scheduling logic and accesses to new data structures like the `kqdom`. However, the cache miss difference is minor in comparison to the other three code paths and has little impact on the overall performance.

CPU affinity has fewer cache misses compared to queue affinity. This is due to connection migration between cores. Recall that the queue affinity policy does not follow the migration. In this workload, the difference is minor enough to not show a significant latency difference.

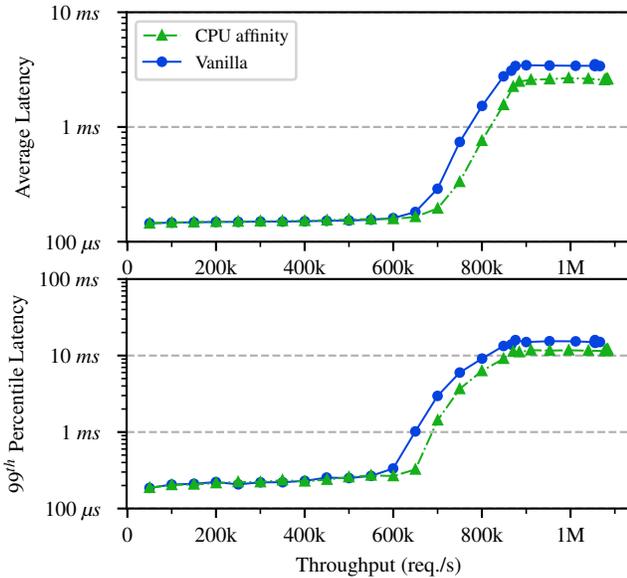


Figure 5: The latency response of Memcached using vanilla and our cache affinity policies. The CPU affinity policy increases low-latency throughput and exhibits lower tail latency in medium to high throughput range.

6.4 Memcached

We benchmark Memcached using Mutilate with the Facebook ETC workload [1]. Figure 5 shows the latency response of the best-performing policy. The largest latency gap occurs at 640k req./s where CPU affinity had $3\times$ lower latency than vanilla. With a maximum tail latency of $320\mu\text{s}$, SKQ effectively increases 9% of low-latency throughput (650k req./s vs. 600k req./s). At high load, SKQ achieves 5% higher throughput and 26% lower tail latency.

Mutilate’s request service time distribution is nearly uniform as it only issues GET and PUT requests. We measure the total time each worker thread spent processing requests in userspace at maximum throughput. The thread with the highest processing time spent 2.8% more than that of the thread with the lowest processing time, indicating that load imbalance is insignificant among worker threads.

With a uniform and balanced workload, cache locality policies offer cache affinity without the scheduling overhead of the load balancing policies. Besides preserving cache locality, there is little that can be done in the kernel event subsystem to improve the performance of Memcached without optimizing the application itself, e.g. reducing lock contention [26].

6.5 Application Server

Server applications often need to service client connections with different latency characteristics, e.g. connections that issue only light requests vs. heavy requests. To investigate the benefit of SKQ on such workloads, we developed a GIS appli-

cation server with a Zipf-like distribution of request service times that models MyBikeRoutes-OSM traffic [39].

In this benchmark, each client issues requests that are either light, medium or heavy computation tasks (approximately $10\mu\text{s}$, $50\mu\text{s}$ and $200\mu\text{s}$). Each client connection is assigned to perform a single task. The overall connection characteristics follow a Zipf-like distribution consisting of 95% light, 4% medium and 1% heavy connections. We collect the latency response of all the policies.

Figure 6 shows the most interesting policies. The hybrid policy (CPU affinity and best of two with work stealing) has 6% lower peak throughput compared to vanilla due to the overhead of scheduling. However, the hybrid policy provides much lower tail latency and more low-latency throughput at mid range. At 460k req./s, the hybrid policy reached the largest gap of $9.9\times$ lower tail latency and $3.4\times$ lower average latency. For a maximum tail latency of $280\mu\text{s}$, the hybrid policy extends the low-latency throughput by $3\times$. It is also worth noting that the hybrid policies show benefit even at low throughput. The analysis is further discussed in § 6.6.

To quantify the imbalances, we again measure the total amount of time each worker thread spent processing events in userspace at 460k req./s (the largest gap) for both vanilla and the hybrid policy. The percent difference between the busiest and the least busy thread is 46.1% for vanilla and 1.4% for the hybrid policy. This shows that the hybrid policy significantly reduced the workload imbalance between threads.

Work Stealing: Figure 6 shows the latency response of best of two with and without work stealing. Although best of two mitigates the imbalance to some extent, work stealing further shifts the curve outward, providing lower tail latency and more low-latency throughput. This is because while best of two balances the threads according to statistics at event activation time, work stealing enables SKQ to react faster and respond immediately to runtime changes and imbalance.

Hybrid Policy: Figure 6 also shows that a hybrid policy of CPU affinity and best of two with work stealing further extends the low latency throughput. This is because CPU affinity also considers cache locality. The hybrid policy always considers the thread with the better cache locality, and prefers it unless best of two shows substantial benefit.

6.6 RocksDB

RocksDB is a fast and persistent embeddable key-value store. Cao et al. [5] characterized and modeled several Facebook workloads that use RocksDB as the backend. We developed a frontend to RocksDB that services GET, SEEK and PUT requests. In this benchmark, we use the same client configuration as the ZippyDB (prefix_dist) model specified by Cao. We place the RocksDB directory on a memory backed file system and preload it with 50M keys.

Unlike Memcached, the ZippyDB workload presents heavy imbalances due to its 3% SEEK requests. As an experiment,

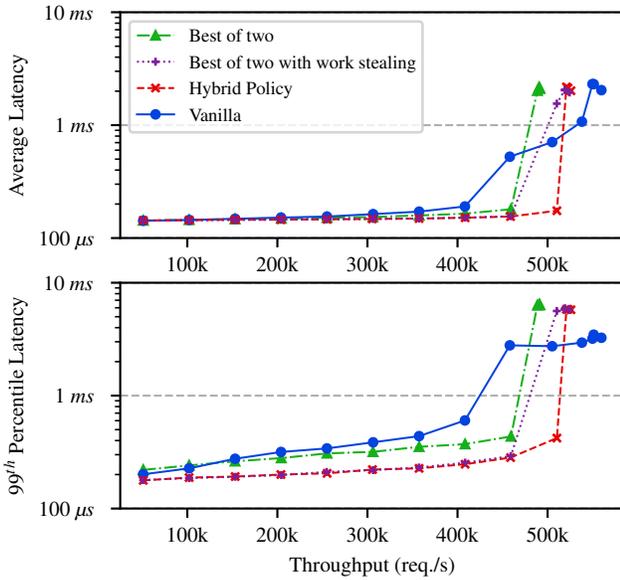


Figure 6: The latency response of the application server using various policies. The hybrid policy refers to CPU affinity and best of two with work stealing. All workload balancing policies lower the latency to different extents.

we eliminate all SEEK requests from the workload and observed $5.85\times$ higher maximum throughput (737k req./s vs. 126k req./s). Figures 7 and 8 show the latency data over full and low throughput range. At low throughput, the tail latency of vanilla starts to increase at only 2.3k req./s whereas the hybrid policy (CPU affinity and best of two with work stealing) stays flat. This indicates extreme imbalances in the workload and is in line with what Cao reported.

Despite a lower peak throughput, the hybrid policy extends the low-latency throughput by $27.4\times$ (63k req./s vs. 2.3k req./s). The largest latency gap occurs at 63k req./s where the hybrid policy achieves $1022\times$ (263 μ s vs. 269 μ s) lower tail latency. This benchmark demonstrates SKQ’s benefit to highly imbalanced workloads.

Unlike the GIS application workload where connections have a fixed request type, RocksDB connections all have the same request service time distribution. At low throughput, heavy SEEK requests are relatively rare where they can still be handled in time and do not queue up. In the GIS application, even a few misassigned heavy connections could overload a thread at low throughput, which is why the hybrid policy showed benefit earlier.

6.7 Web Server

So far we showed the benefit of SKQ on both balanced and imbalanced in-memory workloads. In this benchmark, we use our custom web server to demonstrate how SKQ improves IO-heavy workloads.

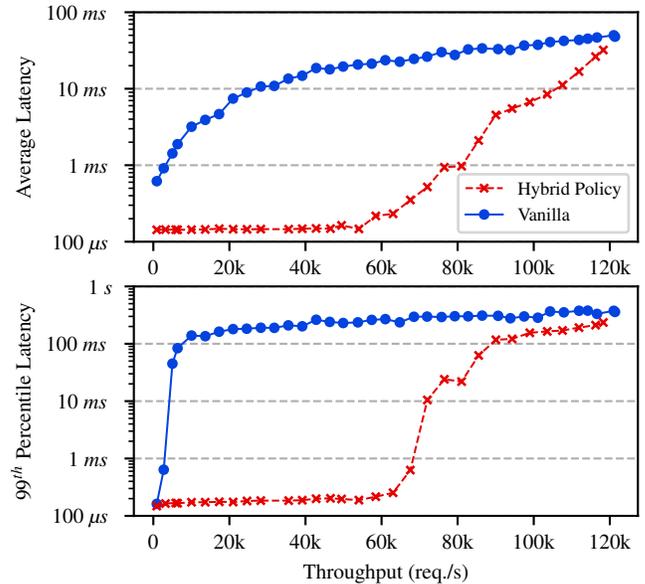


Figure 7: The latency response of RocksDB using the Facebook ZippyDB workload. The hybrid policy refers to CPU affinity and best of two with work stealing. SKQ significantly improves the low-latency throughput.

The web server is configured to service HTTP requests of small HTML pages (~ 640 bytes) without caching residing on a HDD. Figure 9 shows the results of the best-performing policy. Similar to other imbalanced benchmarks, the hybrid policy has marginally lower peak throughput, but provides 19.8% higher low-latency throughput with 300 μ s maximum tail latency. The biggest tail latency improvement occurs at 341k req./s where vanilla has $3.6\times$ higher tail latency (1386 μ s vs 384 μ s). We can see that SKQ also offers benefit on heavy, uniform IO workloads.

In this benchmark, clients request web pages of similar size, which is a uniform IO workload. However, the web server benefits most from a workload balancing policy. This is because IO requests have high variance in latency. Some HTML pages might be in the file system buffer cache whereas others need to be read from the hard disk. This results in different access times and leads to an imbalanced workload. However, compared to our RocksDB and application server benchmark, IO does not induce as much imbalance. Thus, the web server benchmark shows less benefit than the other imbalanced workloads.

6.8 Event Prioritization

This benchmark demonstrates the effect of event prioritization on a lightweight high priority client. We use the same workload distribution as the application server.

The measurement client is assigned regular priority in the first experiment but high priority in the second experiment.

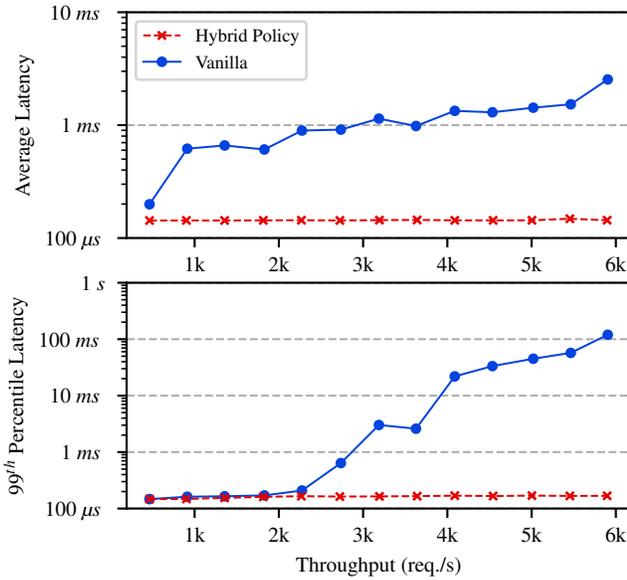


Figure 8: The low throughput latency response of Figure 7. Vanilla’s latency rises at very low throughput.

The measurement client also sends 30000 lightweight requests per second to probe the latency. We set `rtfreq` to be 10000 so that server threads call `kevent()` every 100 μ s. `rtshare` is left at 100 to let SKQ pass back as many high priority events as possible. We measure the latency experienced by the measurement client in both settings and the regular clients’ latency in the second experiment.

Figure 10 shows the latency response. At peak throughput, the high priority client’s throughput comprises 5% of all traffic. The high priority measurement client’s tail latency only increases by $1.2\times$ (371 μ s) than that at low throughput (263 μ s) whereas the regular priority measurement client’s tail latency increases by almost $10\times$ (2656 μ s). In addition, the regular clients did not experience increased latency due to the addition of the high priority client but the server did lose 0.9% maximum throughput. This benchmark shows that event prioritization enables the server to service a lightweight, high priority client with low tail latency while having minimal impact on regular clients even when the server is fully loaded.

The tunables (`rtshare`, `rtfreq`, high priority client’s request rate) used are specifically tuned for this workload. Blindly changing the tunables to meet unrealistic goals only lead to decreased performance. For example, if the request rate of the high priority client is too high, it will experience increased latency as serving a large amount of high priority requests with low latency without starvation is difficult. Furthermore, using inappropriate `rtshare` and `rtfreq` may cause starvation or priority loss. Therefore, we do not offer recommended values for the tunables as they are highly situational.

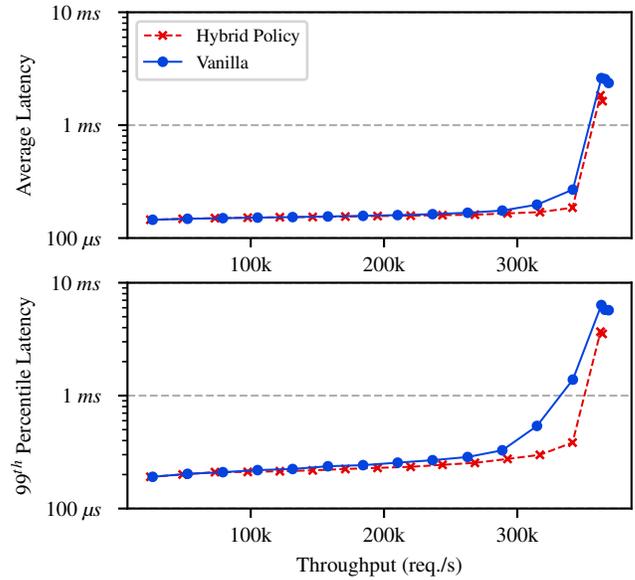


Figure 9: The latency response of our web server with file caching turned off. The hybrid policy refers to CPU affinity and best of two with work stealing. SKQ provides more low-latency throughput for IO-heavy workloads.

6.9 Comparing with a Kernel-bypass System

We compare SKQ with the state-of-the-art kernel-bypass system Shenango to illustrate the benefits and tradeoffs. We also considered porting SKQ to F-Stack, a kernel-bypass library based on FreeBSD’s networking stack. Unfortunately, F-Stack does not support multithreading within a single instance [13]. While outperforming kernel-bypass techniques in a traditional OS is unlikely without significant advances in hardware and software, we can still narrow the performance gap.

We built Shenango on Debian 10 running on a Skylake machine with an Intel 82599 NIC as Shenango only supports two modified NIC drivers in DPDK. We use one Skylake machine as the measuring client and six Skylake machines as workload generation clients. We benchmark three synthetic workloads with different request service time distributions: two with uniform distribution (10 μ s, 20 μ s) and one with imbalanced Zipf-like distribution (85% 10 μ s, 12% 50 μ s, 3% 200 μ s). We compare how each system maximizes the low-latency throughput (<150 μ s 99th percentile latency).

This comparison is unfair due to vastly different approaches and environments. First, Shenango implements a custom networking stack using DPDK and user-level scheduling, both of which bypass the Linux kernel. SKQ only redesigns the event subsystem in FreeBSD. Second, Shenango’s TCP stack is a research prototype and lacks TCP features such as congestion control, while SKQ uses FreeBSD’s networking stack.

In Figure 11, at 10 μ s request service time, Shenango shows $1.67\times$ higher low-latency throughput than SKQ. However,

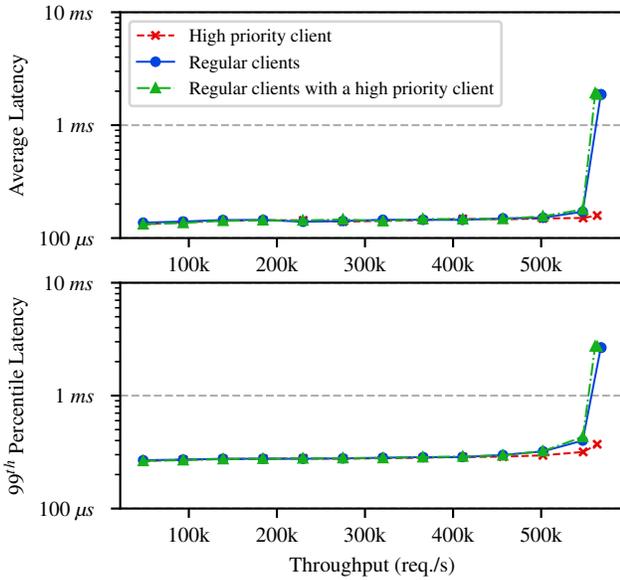


Figure 10: The latency response of two runs where the measurement client is marked as high priority vs. regular priority. The server services a high priority client with low latency while having little impact on other clients at peak throughput.

for 20 μ s request service time, Shenango only achieves $1.5\times$ higher low-latency throughput. This is because system call overhead comprises a larger fraction of time when the request service time is low. Heavier requests mask system call overhead as applications spend more time in userspace.

For the imbalanced workload, Shenango shows less benefit than uniform workloads – only 16.7% higher low-latency throughput than SKQ. This gap is lower because of the longer average service times and correspondingly lower OS overhead. SKQ achieves a result very close to Shenango as we measure nearly no load imbalance (§ 6.5). We also measure the latency response of vanilla, i.e., multiple SKQs, and found that SKQ closes the gap between vanilla and Shenango by 83.7%.

Kernel-bypassing shows bigger improvements when the request service time is low and uniform. For imbalanced workloads and heavier requests, the benefit of kernel-bypassing diminishes compared to kernel event scheduling.

Additionally, Shenango suffers from difficult adoption. First, Shenango requires changes to the threading model, synchronization and networking APIs. Porting complex applications such as RocksDB proved to be challenging. SKQ requires less invasive changes and affects only one system call. As a comparison, Shenango’s Memcached results in ~ 1200 SLOC changed whereas SKQ’s Memcached modifies only ~ 200 SLOC, including SKQ statistics collection and exposing SKQ controls to configuration files. Second, Shenango requires DPDK and patching NIC drivers, which increases maintenance effort. SKQ is built into the kernel and supports all NICs supported by the operating system.

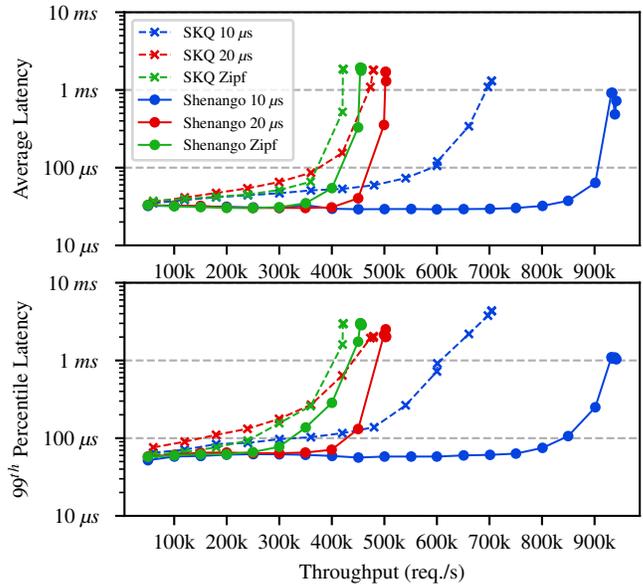


Figure 11: The latency response of three workloads running on Shenango and SKQ. SKQ uses CPU affinity scheduling policy for 10 μ s, 20 μ s benchmarks; CPU affinity and best of two with work stealing for the Zipf-like benchmark.

7 Conclusions

SKQ revisits existing kernel event facilities and provides a practical solution to the latency problem for applications running in traditional OSes. SKQ offers novel features and a production quality implementation that has been developed over the course of nearly two years. We show significant performance gains in applications by only revisiting kernel event subsystems. This suggests that there are still optimization opportunities in traditional OS kernels.

Availability

All of our code and scripts to run experiments are available at <https://rcs.uwaterloo.ca/skq/>.

Acknowledgments

The authors would like to thank Samer Al-Kiswany, Tim Brecht, Ryan Hancock, Martin Karsten, Amilios Tsalapatis and Bernard Wong for fruitful discussions during SKQ’s development. We would also like to thank the anonymous USENIX ATC reviewers for their valuable feedback, especially our shepherd Anton Burtsev. This research is supported by NSERC Discovery grant, Waterloo-Huawei Joint Innovation Lab grant, and NSERC CRD grant.

References

- [1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.
- [2] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Commun. ACM*, 60(4):48–54, March 2017.
- [3] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.
- [4] Robert Benson. The Event Completion Framework for the Solaris Operating System. http://developers.sun.com/solaris/articles/event_completion.html, July 2004.
- [5] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC 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.
- [6] Jonathan Corbet. Receive packet steering. <https://lwn.net/Articles/362339/>, November 2009.
- [7] Jonathan Corbet. Ringing in a new asynchronous I/O API. <https://lwn.net/Articles/776703/>, January 2019.
- [8] Microsoft Corporation. Introduction to Receive Side Scaling. <https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling>, 2017.
- [9] Microsoft Corporation. I/O Completion Ports. <https://docs.microsoft.com/en-us/windows/win32/fileio/i-o-completion-ports>, May 2018.
- [10] Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, Washington, First edition, 1992.
- [11] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Commun. ACM*, 56(2):74–80, February 2013.
- [12] Jake Edge. Receive flow steering. <https://lwn.net/Articles/382428/>, April 2010.
- [13] F-Stack. Can f-stack network stack run as 1 process on multiple cores with multiple threads? <https://github.com/F-Stack/f-stack/issues/27>, June 2017.
- [14] Facebook. RocksDB. <https://rocksdb.org>, 2020.
- [15] FreeBSD Foundation. lockstat – report kernel lock and profiling statistics. <https://www.freebsd.org/cgi/man.cgi?query=lockstat&sektion=1&manpath=freebsd-release-ports>, 2015.
- [16] Google, Inc. The Go Programming Language. <https://golang.org/>, September 2019.
- [17] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12*, pages 135–148, Berkeley, CA, USA, 2012. USENIX Association.
- [18] Intel. Introduction to Intel Ethernet Flow Director and Memcached Performance. <https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf>, 2017.
- [19] Intel. Data Plane Development Kit (DPDK). <https://www.dpdk.org>, 2020.
- [20] Intel Corporation. Performance Testing Application Device Queues (ADQ) with Redis. <https://www.intel.com/content/www/us/en/architecture-and-technology/ethernet/application-device-queues-with-redis-brief.html>, 2019.
- [21] Internet Engineering Task Force. IETF HTTP Working Group. <https://httpwg.org/>, May 2020.
- [22] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, 2014.
- [23] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μ second-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.
- [24] Dan Kegel. The C10K Problem. <http://www.kegel.com/c10k.html>, May 1999.

- [25] Kimberly L. Tripp, Conor Cunningham, Adam Machanic, Ben Nevarez, Kalen Delaney, Paul S. Randal. *Microsoft SQL Server 2008 Internals*. Microsoft Press, Redmond, Washington, 1 edition, 2008.
- [26] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 9. ACM, 2012.
- [27] Jonathan Lemon. Kqueue: A Generic and Scalable Event Notification Facility. In *USENIX Annual Technical Conference, FREENIX Track, ATC '01*, Berkeley, CA, USA, 2001. USENIX Association.
- [28] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing, SOCC '14*, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.
- [29] Linux Man Page Project. `epoll` - I/O event notification facility. <http://man7.org/linux/man-pages/man7/epoll.7.html>, September 2019.
- [30] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkupati, William C Evans, Steve Gribble, et al. Snap: a Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 399–413, 2019.
- [31] Michael Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, October 2001.
- [32] Nick Mattewson, Azat Khuzhin, and Niels Provos. libevent - an event notification library. <https://libevent.org>, September 2019.
- [33] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.
- [34] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12*, pages 337–350, New York, NY, USA, 2012. ACM.
- [35] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):1–30, 2015.
- [36] 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.
- [37] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware Thread Management. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'18*, pages 145–160, Berkeley, CA, USA, 2018. USENIX Association.
- [38] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 2*. Microsoft Press, Redmond, Washington, Sixth edition, 2012.
- [39] Yasir Shoaib and Olivia Das. Web Application Performance Modeling Using Layered Queueing Networks. *Electronic Notes in Theoretical Computer Science*, 275:123–142, 2011.
- [40] Tencent Cloud. F-Stack | High Performance Network Framework Based On DPDK. <http://www.f-stack.org/>, 2020.
- [41] The Open Group. `lio_listio` - list directed I/O. https://pubs.opengroup.org/onlinepubs/9699919799/functions/lio_listio.html, 2017.

A Linux Kernel Implementation of the Homa Transport Protocol

John Ousterhout
Stanford University

Abstract

Homa/Linux is a Linux kernel module that implements the Homa transport protocol. Measurements of Homa/Linux reconfirm Homa's superior performance compared to TCP and DCTCP. In a cluster benchmark with 40 nodes, Homa/Linux provided lower latency than both TCP and DCTCP for all message sizes; for short messages, Homa's 99th percentile tail latency was 7–83x lower than TCP and DCTCP. The benchmarks also show that Homa has eliminated network congestion as a significant performance limitation. Both tail latency and throughput are now limited by software overheads, particularly software congestion caused by imperfect load balancing of the protocol stack across cores. Another factor of 5–10x in performance can be achieved if software overheads can be eliminated in the future.

1 Introduction

Montazeri et al. recently introduced a new network transport protocol for datacenters called Homa [25]. Homa uses network priority queues and receiver-driven packet scheduling to favor shorter messages and eliminate congestion at host downlinks. Homa reduces latency significantly, especially for short messages under high network loads. Montazeri et al. demonstrated tail latencies almost 100x better than the best prior measurements of transport protocols that included TCP, DCTCP [1], Infiniband, HULL [2], PIAS [4], and NDP [15].

However, the original evaluation of Homa was done in a research setting, based partly on simulations and partly on a user-level implementation (with kernel bypass) in the RAM-Cloud storage system [29]. The implementation did not support general applications, and it was difficult to compare it fairly to a kernel TCP implementation, since the user-level implementation of Homa avoided the high software overheads imposed on TCP by the kernel. As a result, the original Homa work left open questions about whether the protocol's benefits could be achieved in a more practical setting.

This paper describes Homa/Linux, a Linux kernel module that implements Homa. I undertook the development of Homa/Linux with three goals: first, to understand how the overheads of a kernel implementation affect Homa's implementation and performance; second, to measure how a kernel version of Homa performs compared to a mature and widely used implementation of TCP and DCTCP; and third, to create a practical implementation of Homa to encourage its use in real applications and evaluate its benefits for production datacenter workloads.

This paper makes two contributions. First, Homa/Linux demonstrates that it is possible to build a competitive production implementation of Homa. Homa/Linux outperforms both TCP and DCTCP by a wide margin, confirming the results in [25]. Using four of the Montazeri workloads and considering short messages under high network loads, P99 (99th percentile) latency under Homa/Linux is 7–83x lower than TCP and DCTCP. All message sizes under all workloads experience lower latency with Homa than either TCP or DCTCP, both at the median at P99. In most cases Homa's P99 latency is lower than the median for TCP and DCTCP. Homa needs only a few priority levels to achieve high performance, and it outperforms TCP and DCTCP even with only one priority level.

Second, this work provides a case study of the challenges in building a performant transport protocol in the high-overhead environment of the Linux kernel. Homa/Linux had to address a variety of issues, such as batching, load balancing, and real-time processing. Homa eliminates almost all network congestion, but software congestion is becoming more problematic as more and more cores must be harnessed to keep up with increasing network speeds. In Homa/Linux, software overheads are now the primary factor limiting performance. This paper quantifies those overheads. For example, at least 18 cores will be required to drive a 100 Gbps network in both directions, and distributing protocol processing across multiple cores increases software overheads by 2–3x.

Although Homa/Linux provides much better performance than TCP, its latency and small-message throughput are still 5–10x worse than raw network speeds. Section 7 argues that this gap cannot be reduced significantly as long as transports are implemented in software. To harness future networks' full performance potential, protocols such as Homa will probably need to be implemented in hardware. This will require new NIC architectures to be developed.

2 Homa Summary

This section summarizes the key elements of the Homa protocol; see Montazeri et al. [25] for details and discussion. Homa is designed as a transport for RPC frameworks in datacenters. It is optimized for networks with one-way hardware latencies as low as 1–2 μ s and aims to provide the lowest possible tail latency for short messages, even when operating at high network load with a mix of message lengths. It does so by minimizing the latency impact of congestion at the network edge (host downlinks). Homa also eliminates head-of-line block-

ing that occurs when small messages are delayed behind large ones in TCP streams. Homa does not explicitly deal with congestion in the network core (see Section 8).

SRPT and messages. Homa implements an approximation of SRPT (shortest remaining processing time), prioritizing shorter messages over longer ones. SRPT is most beneficial for short messages, but it also improves latency for large messages compared to the fair sharing approach used in protocols such as TCP. This is because SRPT produces run-to-completion behavior: once a message becomes highest priority, it will remain highest priority until it completes (unless new messages arrive).

Receiver-driven packet scheduling. In Homa, each receiver collects information about all of its incoming messages and schedules incoming packet transmissions. The first few packets of each message (*unscheduled packets*) are transmitted unilaterally by the sender, but the remaining *scheduled packets* are sent only in response to *grants* from the receiver. The number of unscheduled packets is typically chosen to cover the round-trip time, so that an unloaded server can return the first grant before the last unscheduled packet has been sent. This allows messages to use the full network bandwidth in an unloaded system. The grant mechanism allows receivers to limit congestion at their downlinks; buffer buildup occurs only if many senders transmit unscheduled packets simultaneously to the same destination (*incast*). I use the term *RTTbytes* to describe the number of unscheduled bytes, following Montazeri et al.; its role is analogous to windows in other protocols.

In-network priority queues. Homa takes advantage of the priority queues in modern network switches (typically 8 or 16 for each egress port). It divides the priority levels into two groups: the highest levels are used for unscheduled packets, and the lower levels for scheduled ones. Within each group, shorter messages get higher priorities. Receivers choose the priorities for all of their incoming traffic. For scheduled packets, the receiver specifies priorities “just in time” using grants. For unscheduled packets, the receiver specifies in advance the range of message lengths for each priority level; it disseminates new assignments occasionally as its workload changes.

The fraction of priority levels allocated for unscheduled packets is chosen to match the fraction of all incoming bytes that are in unscheduled packets. The cutoffs between unscheduled priorities are assigned so that each priority level is used for about the same number of incoming bytes.

Sender-side SRPT. Homa nodes also implement SRPT when transmitting: given multiple messages with packets to transmit, a sender should transmit packets for the message with the fewest remaining bytes. To do this, Homa must limit the rate at which outgoing packets are passed to the NIC to ensure that long queues do not build up in the NIC (these would delay packets from new shorter messages).

Overcommitment. When a receiver issues a grant there is no guarantee that the sender will immediately transmit the granted packets. In order to keep its downlink fully loaded,

receivers thus grant simultaneously to multiple incoming messages; this is called *overcommitment* since it overcommits the downlink and may result in buffering at the switch. Scheduled priority levels are allocated to ensure SRPT among the messages being granted, so overcommitment does not affect the latency of the highest priority message. The degree of overcommitment is typically in the range of 5–10, which is chosen as a balance between excessive buffer buildup (if too large) and inefficient use of downlink bandwidth (if too small).

3 Homa/Linux API

TCP’s connection-oriented socket API is a poor match for Homa, and for datacenter applications in general. The first problem is TCP’s streaming nature, which means that it has no notion of message boundaries. This is problematic for Homa, which needs message lengths to implement SRPT. Streams are also awkward for most datacenter applications, which communicate via messages. These applications must impose their own message structure on TCP streams; this adds nontrivial complexity, given that a read might return only part of a message, or parts of several messages. It is difficult to share a TCP stream between multiple reading threads (e.g., in a server), since reads don’t necessarily return dispatchable units (entire messages).

Streams also have the disadvantage of enforcing FIFO ordering on their messages. As a result, long messages can severely delay short ones that follow them. This head-of-line blocking is one of the primary sources of tail latency measured for TCP in Section 5.

A second problem with TCP sockets is that they are connection oriented, with long-lived state for each peer that an application communicates with. Connections are undesirable in datacenters because applications can have hundreds or thousands of them, resulting in high space and time overheads. Some applications have resorted to proxies or other forms of connection pooling [32] to reduce the overheads. It seems to be an article of faith in the networking community that connections are necessary for desirable properties such as reliable delivery and congestion control, but in fact all of these properties can be achieved without connections.

Because of these issues, Homa/Linux provides a message-based API. Specifically, it implements *remote procedure calls* (RPCs). Each RPC consists of a *request* message sent from a client to a server, followed by a corresponding *response* message in return. An RPC protocol has two advantages over a pure messaging approach. First, the response serves as acknowledgment for the request, reducing the number of packets that must be processed. Second, it results in a deeper interface [28] because the transport can implement the timers and retries needed to ensure end-to-end completion. In a pure messaging protocol, timeouts must be implemented by applications (the protocol can ensure that individual messages are delivered, but cannot ensure that servers generate responses).

Homa/Linux has no notion of connections, only RPCs. A client can use a single socket for simultaneous communication

```

int socket(AF_INET, SOCK_DGRAM, IPPROTO_HOMA);
int bind(int sockfd, const struct sockaddr *addr,
        socklen_t addrlen);
int close(int sockfd);

int homa_send(int sockfd, const void *request,
             size_t reqlen,
             const struct sockaddr *dest_addr,
             socklen_t addrlen, uint64_t *id);
int homa_recv(int sockfd, void *buf, size_t len,
             int flags, struct sockaddr *src_addr,
             socklen_t addrlen, uint64_t *id);
int homa_reply(int sockfd, const void *response,
             size_t resplen,
             const struct sockaddr *dest_addr,
             socklen_t addrlen, uint64_t id);

```

Figure 1: The API provided by Homa for applications; `socket`, `bind`, and `close` are existing Linux system calls.

with any number of servers. A server can use a single socket to receive requests from any number of clients, to reply to client requests, and to issue its own requests as a client. Homa/Linux maintains state only for active RPCs (rarely more than a few at a time). This eliminates the overheads associated with connections.

Each Homa RPC is independent. Any number of RPCs may be active simultaneously for a given socket. If a client issues multiple concurrent RPCs, they may complete in any order; this eliminates the TCP's head-of-line blocking problem. If order matters among concurrent RPCs, Homa clients can add their own sequence numbers to messages.

Figure 1 shows Homa/Linux's system call interface. The existing `socket` and `close` calls are used to create and delete Homa sockets; Homa/Linux defines a new protocol type `IPPROTO_HOMA`. If a socket is going to receive incoming requests, the existing `bind` call is used to associate the socket with a well-known port number. There is no need to invoke `bind` for client sockets; Homa/Linux automatically assigns port numbers for them from the upper half of the 16-bit port space.

Homa/Linux defines three new functions: `homa_send`, `homa_recv`, and `homa_reply`. All are implemented via `ioctl`s on the Homa socket; Homa/Linux does not add new system calls to Linux. A client invokes `homa_send` to issue a request; it returns a 64-bit unique identifier for the RPC, which can be used to wait for the corresponding response. `homa_recv` receives incoming messages; it returns a message and its unique identifier. The arguments to `homa_recv` can restrict it to return only requests, only responses, or only a response with a given identifier. `homa_reply` is used by servers to send responses; it is similar to `homa_send` except that the RPC identifier is an input argument.

The Homa/Linux API is not backwards-compatible with existing applications based on TCP or UDP sockets, due to the API's need for explicit RPC ids. However, most datacenter applications are layered on a few RPC frameworks such as gRPC [12] or Apache Thrift [34]; adding Homa support to them should enable transparent usage by many applications.

Homa/Linux ensures reliable delivery of messages and

eventual completion of each RPC issued by a client. An RPC fails only if the server becomes nonresponsive or there is no socket matching the port in the request. There is no limit on the processing time for an RPC.

4 Implementation

Homa/Linux is a dynamically loadable Linux kernel module; it does not require any changes to Linux itself. The current version runs on Linux version 5.4.80 and contains about 10,000 lines of heavily commented C code. Source for Homa/Linux is freely available on GitHub along with unit tests, instrumentation tools, and all of the benchmarks discussed in this paper [16]. The module was implemented with the goal of achieving production quality, and I believe the current version is mature enough to run a variety of applications.

Most of the implementation effort for Homa/Linux focused on three obstacles to high performance. The first is the high cost of pushing a packet through the protocol stack. This cost is particularly high in Linux, due to its many layers and features. To amortize the protocol stack overheads, packets must be collected into batches, so that the stack traversal cost is only incurred once per batch. Unfortunately, batching necessarily incurs latency, since the first packet in a batch must be delayed until the last is available. Larger batches can be processed more efficiently, but incur a larger latency penalty.

The second obstacle is that a single core is too slow to handle all the protocol processing for a modern high-speed network, especially for small packets. This problem is worsening over time, since network speeds are increasing rapidly while CPU speeds are not. A transport protocol must be able to balance its load across a large number of cores. Even so, small-packet performance will be severely limited by software overheads; for example, neither Homa nor TCP can utilize more than about one-third of the bandwidth of a 25 Gbps network for short-message workloads, even with 20 cores. The greatest challenge with load balancing is software congestion in the form of hot spots that form when too much work is assigned to one core; even after considerable optimization, hot spots remain the single greatest source of tail latency for Homa/Linux.

The third obstacle to high performance is real-time processing. In the case of Homa, real-time processing is needed to implement a transmit rate limiter as discussed in Section 4.3. Implementing such a mechanism, which requires response times on the order of a few microseconds, is challenging in the Linux kernel and creates additional software overheads.

These problems are likely to occur in any transport protocol, and Linux contains mechanisms to facilitate both batching and load-balancing. However, the Linux mechanisms are biased towards TCP, even when implemented in code that is ostensibly protocol independent. To implement Homa efficiently, it was necessary to reshape Homa/Linux to fit the existing mechanisms. In some cases, Homa/Linux subverts the mechanisms to achieve results that are not possible with TCP.

One recurring issue is that TCP depends on in-order processing of a stream's packets. This requires flow-consistent

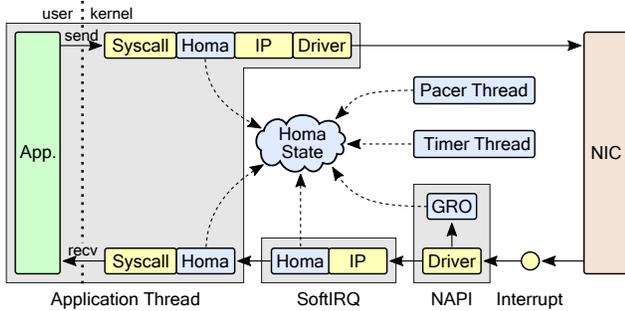


Figure 2: Structure of Homa/Linux. Homa components are shown in blue; existing Linux kernel modules are in yellow. Gray areas represent different cores. Only the primary sending and receiving paths are shown; other Homa elements such as the pacer thread and timer thread also transmit packets.

routing in the network, and also requires consistency in protocol processing on the hosts: for example, to preserve packet order, each packet of a flow must pass through the same set of cores. Homa does not require packets to be processed in order; by relaxing this constraint it can achieve better batching and load balancing. As a result, Homa/Linux attempts to bypass TCP’s constraints; this is not always easy or clean, since the constraints tend to be enforced by Linux.

The rest of the section discusses these three primary issues in detail, plus a few other smaller issues. The design described here evolved over about two years and was driven by extensive performance measurements. For example, throughput for Homa/Linux improved by about 3x over this process (the initial version did not implement batching). Space limitations prevent inclusion of the detailed measurements.

4.1 Packet flow and batching

Figure 2 shows the major components of Homa and the basic packet flow for sending and receiving messages. Homa exists as a layer just above IP in the Linux networking stack, parallel to TCP and UDP; the packet flow discussed in this section is similar to that for TCP and UDP. Homa packets are encapsulated as IPv4 datagrams.

The basic path for packet transmission is shown in the top part of Figure 2. When an application invokes `homa_send` or `homa_reply`, the resulting `ioctl` kernel call is dispatched to Homa/Linux. Homa/Linux copies the message data from user space into packet buffers and passes the first few packets to the IP stack, which eventually calls a device driver to transmit the packets.

Linux offers two mechanisms for batching on the transmit path: TSO (TCP Segmentation Offload) and GSO (Generic Segmentation Offload). In either case, the transport protocol creates a single large packet (up to 64 KB) for stack traversal. Once the packet reaches the device driver it is broken up into smaller segments for transmission. With TSO the segmentation occurs in the NIC. For NICs that don’t support TSO, or for protocols that cannot use TSO, GSO can perform segmentation in software, just above the driver layer. Homa/Linux currently supports TSO but not GSO. TSO assumes that packets have TCP headers, so Homa packet headers mimic the

TCP fields that TSO depends on. Unfortunately, TSO and GSO enable batching only for data packets that are part of the same message. For short messages and control packets, each transmitted packet must traverse the protocol stack independently.

The receive path is shown at the bottom of Figure 2. When an interrupt occurs, the interrupt handler doesn’t actually read packets; instead, it schedules an *NAPI* action to execute driver code. NAPI actions execute at interrupt level on the same core that received the interrupt, just before returning from the interrupt and with interrupts reenabled. To avoid high overheads associated with interrupts, the NAPI layer polls the NIC for more packets until it reaches either a packet count limit or a time limit.

Once the NAPI layer has finished collecting packets, it passes them to the *SoftIRQ* layer, where they work their way through the networking and IP layers, eventually reaching Homa. Homa performs protocol processing and message re-assembly. When a message is complete, Homa signals the waiting application thread (or queues the message if there are no waiting threads); the application thread copies the message data to user space and then returns from the `homa_recv` system call. *SoftIRQ* code executes in the same fashion as NAPI (at interrupt level with interrupts enabled) but it may run on a different core than the NAPI code (see Section 4.2).

Before forwarding incoming packets to *SoftIRQ*, the NAPI layer organizes them into batches. It does this by passing each packet to a transport-specific function using a mechanism called GRO (Generic Receive Offload). The transport-specific GRO function decides how to group packets into batches. For TCP, each batch contains packets from a single flow (this ensures that all packets from a flow go to the same core for *SoftIRQ*). Unfortunately, protocol-independent GRO code partially segregates batches by flow before invoking the transport-specific function. This segregation is neither necessary nor desirable for Homa. Fortunately, I found a way for Homa/Linux to defeat the GRO segregation and collect all incoming packets into a single batch (see the Appendix for details). This allows Homa to batch more aggressively than TCP, and is one of the reasons Homa has higher throughput for short messages (in TCP, each short message becomes a separate batch).

4.2 Load balancing

As mentioned in the introduction to this section, the networking subsystem must distribute load across multiple cores to keep up with a high-speed network. Load balancing is easy for packet output, because the output stack executes entirely on the sending thread’s core, with a separate NIC channel per core. The Linux scheduler balances threads across cores, and this distributes the packet transmission load as well.

The remainder of this section focuses on packet input, where Linux employs two different load balancers. The first is RSS, which runs in NICs and distributes incoming packets across cores for NAPI processing using a hash of packet

header fields. This hash function ensures that all packets from a given flow are assigned to the same core, as required by TCP. The second load balancer runs in the NAPI layer: after incoming packets have been grouped into batches, the NAPI layer chooses a SoftIRQ core for each batch. It does this using another hash function on header fields of the first packet in each batch; the hash is different from the one used by the NIC, but it ensures that all batches for a given flow are delegated to the same SoftIRQ core.

Input load balancing is problematic because it results in hot spots. Hot spots occur because the Linux scheduler's load balancer is unaware of the networking subsystem. For example, the NAPI layer can consume a core for hundreds of microseconds when a burst of packets arrives for a single long message. Short messages can still be processed through the NAPI and SoftIRQ layers on other cores, but if a message's target thread is assigned to the core processing the local burst, it will not run until after the burst is handled. This form of hot spot is the primary contributor to P99 tail latency in Homa/Linux.

Homa exacerbates this form of hot spot because it tries to use the entire incoming link bandwidth for one message at a time. TCP's fair-sharing scheduler tends to divide incoming bandwidth among multiple flows, which will probably be processed on different cores, so this form of hot spot is less likely to occur. Increasing packet size does not help, because NAPI polls for more packets: if the MTU increases then NAPI will spend less time processing packets but more time polling.

Homa/Linux has been unable to eliminate NAPI-thread conflicts because it has no control over the RSS hash function in the NICs. Hot spots can also occur with the SoftIRQ layer, where it conflicts with either application threads or NAPI. I eliminated most of these hot spots by repurposing an alternate mechanism for SoftIRQ core selection (the socket flow table, which steers packets to a specific application thread's core) so that Homa's GRO function can select a specific SoftIRQ core for each packet batch. After experimenting with several policies, I settled on one that records for each core the most recent time when it processed a Homa packet at either NAPI or SoftIRQ level. To choose a SoftIRQ core, NAPI consults the times for the next few cores after the one where it is running (in circular order) and selects the one with the oldest time. This policy has two desirable properties. First, it avoids conflicts between NAPI and SoftIRQ. Second, it tends to distribute SoftIRQ batches across multiple cores, so that no single core is occupied continuously: this leaves time for application threads to execute on each core. This policy improved Homa's P99 latency by about a factor of 2x.

I experimented with using core affinity to reduce conflicts between application threads and NAPI/SoftIRQ threads. For example, I tried partitioning the cores, with one set used exclusively for application threads and the others for NAPI and SoftIRQ processing. However, I was unable to find a configuration where core affinity improved tail latency.

A second problem with load balancing is that it hurts performance at low load. At low load it is best to concentrate

all processing on a single core: ideally, RSS should be disabled so that all interrupts pass through a single core, and that same core should execute both the NAPI and SoftIRQ layers. This maximizes cache locality, eliminates inter-core synchronization, eliminates the overhead of cross-core invocation between the NAPI and SoftIRQ layers, and makes the NAPI polling mechanism more efficient. In contrast, load balancing employs many cores but at low load they are all underutilized. These cores are likely to enter power-saving *C-states*, resulting in wakeup delays of 50–100 μ s the next time work is assigned to them. Furthermore, the continual arrival of packets on all cores reduces the effectiveness of power-saving mechanisms.

Ideally, the load balancing configuration should change dynamically with load, but there does not appear to be a way to do this in Linux. Thus, Homa/Linux is optimized for high loads. As a result, P99 latency for short messages is actually worse at low load than high load (see Section 5.4). However, Homa/Linux includes one optimization for low load. The algorithm for choosing a SoftIRQ core checks to see if the current batch contains only a single packet (an indicator of low load). If so, the NAPI core is also used for SoftIRQ. This reduces round-trip latency by 3–4 μ s when the system is underloaded.

4.3 Real-time processing: the pacer

As discussed in Section 2, Homa/Linux must limit the length of the NIC's internal transmit queue to enforce SRPT during output: if a large queue builds up in the NIC, then it will delay small messages. When the NIC queue length exceeds a threshold, Homa/Linux no longer transmits packets immediately; their messages are added to a queue and a *pacer* thread is awakened. The pacer thread manages the queue, passing packets to the NIC in SRPT order as NIC queue length permits.

The NIC queue length is not directly observable, so Homa/Linux maintains `nic_empty_time`, an estimate of the time when all queued packets will have been transmitted. As each transmitted packet is passed to the IP stack, this variable is updated based on the packet length and link speed. A system parameter controls how far into the future `nic_empty_time` is allowed to get; the current value is 2 μ s.

The pacer operates under severe real-time constraints: to keep the NIC queue short while fully utilizing the uplink, it must queue new packets at a time granularity of 1–2 μ s. There is no sleep/wakeup mechanism in Linux that can operate at this timescale, so the pacer thread must poll the processor cycle counter to wait for the next transmission time. As a result, the pacer consumes most of a core under heavy output load.

Even with a dedicated core, the pacer can fall behind. In the original design, once the pacer queue became nonempty, all output packets passed through the pacer. Unfortunately, a single thread cannot push small packets through the IP stack fast enough to drive the network at full speed. In addition, the scheduler occasionally deschedules the pacer; this can produce gaps of several milliseconds with no packets transmitted.

Homa/Linux includes three additional mechanisms to ensure full usage of the uplink. First, packets smaller than a threshold (currently 1000 bytes) bypass the pacer mechanism: they are passed to the NIC immediately without consulting or updating `nic_empty_time` (short packets transmit so quickly that it isn't possible to queue them faster than the NIC sends them, so the pacer is superfluous). Second, other cores help push packets through the network stack when the pacer falls behind. Before queuing a message for the pacer, Homa/Linux checks `nic_empty_time`; if it has dropped below the threshold for sending more packets, it indicates that the pacer isn't keeping the uplink fully utilized, so packets are transmitted directly, even if the pacer queue is occupied. Third, the pacer is invoked explicitly by the SoftIRQ layer after processing each batch of Homa packets; if the pacer thread has fallen behind, this invocation will queue more packets (but it returns without polling).

4.4 Grants

The goal of the grant mechanism is to maintain RTTbytes of outstanding grants (data granted but not yet received) for each *active message*. The number of active messages is limited by the degree of overcommitment; only the highest priority messages (according to SRPT) are active. In addition, Homa/Linux will not grant to more than one message from a given peer at a time, since the peer will only transmit the shortest one. To implement this, Homa/Linux maintains a global 2-level priority queue of incoming messages, consisting of a list of messages from each peer, plus a list of all peers with non-empty lists. Each is sorted in increasing order of the number of bytes not yet granted; the active messages consist of the first message from each of the first few peer lists.

Sending grants is not triggered by a clock, but rather by packet arrivals: after the SoftIRQ layer processes each batch of incoming packets, it checks to see if any active messages need additional grants. Packet arrivals may also change the structure of the grant queues (a new message may appear or an existing message may become fully granted).

The original design of Homa called for one grant packet to be sent for each scheduled data packet. However, this approach results in high overheads. Furthermore, the use of TSO means that packets are transmitted in groups, so there is no benefit in sending grants at per-packet granularity. Thus, Homa/Linux provides a parameter that specifies a minimum increment for grants; it is currently set to 10000 bytes, which is the same as the size of Homa's TSO packets.

4.5 Other implementation issues

Locking. The Linux networking stack is designed around per-socket locks. This approach works well for TCP because each flow is associated with its own socket; thus different flows can be processed concurrently without lock conflicts. However, a Homa application typically only has one socket, which is used for all messages. Homa initially used socket-level locking, but this resulted in severe contention for socket locks.

Homa was eventually refactored to use RPC-level locks as the primary synchronization mechanism. However, per-RPC locks created additional complications. For example, when the first packet arrives for an RPC, an entry must be created in a hash table of RPCs associated with the socket; future packets must use the existing RPC structure. However, multiple "first" packets for an RPC might be processed concurrently on different cores; the natural way to synchronize them is to use the socket's lock. To process packets without acquiring the socket lock, Homa/Linux associates a lock with each bucket in the socket's hash table; this lock covers all RPCs in that bucket. Bucket locks synchronize the lookup of an existing RPC with the creation of a new one, while allowing RPCs in other buckets to be processed concurrently.

Another issue with locking is the potential for deadlock. There are several places in Homa/Linux where multiple locks must be held. RPC locks are normally acquired before any others, but there are a few places where an RPC lock needs to be acquired while holding a different lock; this risks deadlock. Code had to be reorganized on a case-by-case basis to prevent deadlock. For example, in some situations the RPC lock can be acquired conditionally; if it is not available, then the operation can be deferred until later.

Combining FIFO with SRPT. Homa's SRPT priority mechanism is ideal for reducing small-message tail latency, and it also works well for most large messages because of its run-to-completion nature. However, under high load, a few of the largest messages may suffer very high tail latency. To mitigate this problem, Homa/Linux directs a small configurable fraction of the total network bandwidth to the *oldest* message instead of using SRPT; this occurs both when issuing grants for incoming messages and in the pacer for outgoing messages. The current fraction is 5%, which eliminates most of the penalty for the largest messages without affecting tail latency for short messages. This improves tail latency for the largest messages by about 2x in pathological cases.

Reaping RPCs. Any long-running operation creates a threat to tail latency because it may delay other work. One example is RPC reaping, which frees the resources for an RPC after it completes. For clients, this is just before returning from `homa_recv`; for servers, it is after the response has been sent. Reaping can take tens of microseconds for large RPCs, most of which is spent freeing packet buffers. Reaping was originally done immediately when RPCs were freed, but it had a noticeable impact on tail latency. On servers, for example, reaping could occur in the SoftIRQ layer while handling an incoming packet, causing unpredictable delays for subsequent packets.

To reduce the impact of reaping on tail latency, Homa/Linux now defers reaping so that it does not occur when an RPC is freed. Instead, reaping occurs in `homa_recv` while waiting for incoming messages: large messages are reaped incrementally in chunks of a few packet buffers, checking for incoming messages after each chunk. This approach

hides the cost of reaping unless the system is so loaded that `homa_recv` never waits. If the pool of unreaped messages grows too large, then `homa_recv` will stop and reap despite the waiting messages; this situation is rare in practice.

Receive polling. If a thread blocks in `homa_recv`, it takes about 2.5 μ s to wake it on message arrival; this contributes significantly to latency. To avoid this cost, Homa polls briefly in `homa_recv` before blocking the thread. This saves about 2 μ s if a message arrives during the polling interval. The interval is a system parameter, currently 50 μ s. Polling is useful primarily when the system is lightly loaded; it has little impact on the tail latency measurements in Section 5.

Timer thread. Homa contains a dedicated in-kernel thread that wakes up at 1 ms intervals, checks for overdue packets, requests retransmissions, and eventually declares a peer dead if it fails to respond. Lost packets are rare, so slow response is usually due to overload on the peer; to minimize the extra load from retransmission requests, the timer thread will only request retransmission for a single RPC at a time for each peer (the oldest one). If a peer is declared dead, then all RPCs to/from that peer are aborted.

Computing unscheduled priorities. Each Homa host must send information to its peers about the priorities to use for unscheduled packets. Priorities are computed by a user-level daemon, `homa_prio`, which runs regularly (currently every 500 ms) and collects information from Homa/Linux about the size distribution for recently received messages. It then decides how many priorities to use for unscheduled packets and computes cutoff values (the largest message size for each priority level) as described in Section 2. If the cutoffs have changed significantly, `homa_prio` passes them to Homa/Linux using the `sysctl` mechanism. Homa/Linux does not immediately notify all its peers; it waits until the next message from each peer, and then sends the new cutoffs.

Long-term state. As mentioned previously, Homa stores no long-term connection information. It does, however, keep long-term state for each peer that it has communicated with. A peer’s state is about 200 bytes, of which almost half consists of cached routing information. The remainder includes information about unscheduled priorities for that peer and information for detecting timeouts and managing grants. This is far less than the 2000 bytes that TCP stores for each open socket. Peer state is created on demand and never discarded.

Configuration parameters. One disadvantage of Homa/Linux is that it has about 30 configuration parameters. Many were added solely for evaluating the implementation and need not be considered in practice. For many others the exact value has little impact on system performance. About 5–10 parameters can have a significant impact on performance; for them the best value depends on hardware characteristics such as network speed, CPU speed, and number of cores. I believe that it is possible to compute their values automatically by running benchmarks, but I leave this to future work. The only configuration parameter

CPU: E5-2640v4 (10 cores, 2.4 GHz)
RAM: 4x 16 GB DDR4-2400 DIMMs
Disk: Intel DC S3520 SSD (480 GB 6G SATA)
NIC: Mellanox ConnectX-4 25 Gbps
Switch: Mellanox 2410

Table 1: CloudLab [10] x1170 hardware configuration used for benchmarking. All nodes ran Linux 5.4.80. Hyperthreads were enabled (2 hyperthreads per core). TSO and RSS were enabled for all protocols, with separate transmit/receive channels for each hyperthread. C-States were enabled, Meltdown mitigations were disabled, and interrupt moderation was disabled. All 40 nodes were connected to a single switch.

	Homa	TCP	DCTCP
100B latency (μ s)	15.1	23.4	24.1
500KB throughput (Gbps)	10.0	20.3	20.5
Client throughput (Gbps)	23.8	23.9	21.4
Server throughput (Gbps)	23.7	23.6	22.4
Client RPC rate (Mops/sec)	1.6	1.0	1.0
Server RPC rate (Mops/sec)	1.6	1.0	1.0

Table 2: Basic Homa and TCP performance. The top two lines used a single client thread issuing back-to-back requests to a single server. Latency was measured end-to-end at application level with 100-byte requests and responses; throughput was measured with 500 KB requests and responses. For the remaining measurements each client had multiple threads; each thread issued multiple concurrent RPCs. Client performance was measured with a single client node spreading requests across 9 server nodes; server performance was measured with 9 client nodes all issuing requests to a single server node. Throughput was measured with 500 KB requests and responses and counts only message payloads; RPC rate was measured with 100-byte requests and responses. Each number represents the best average across five 5-second runs.

for which the best value might vary from application to application is the polling interval; this should probably be made an argument of the `homa_recv` kernel call.

5 Performance Evaluation

This section evaluates Homa/Linux’s performance, comparing it with the Linux implementations of TCP and DCTCP. The most important metric is tail latency for short messages under high load, but the benchmarks also measure performance for large messages and lower loads. This section also evaluates concerns that have been raised in recent papers about Homa’s use of priorities and network buffers.

The benchmarks ran on a 40-node cluster described in Table 1. Each server had 10 cores with two-way hyperthreading enabled. Unless otherwise indicated below, the term “core” refers to a single hyperthread. Unless otherwise stated, all measurements used a maximum packet size (MTU) of 3000 bytes, which produced better results for all protocols than the traditional 1500 bytes. For DCTCP the ECN marking threshold was 70 KB.

5.1 Basic latency and throughput

Table 2 shows latency and throughput for Homa/Linux and TCP under best-case conditions. Homa/Linux round-trip-times for short messages are about one-third lower than those of TCP or DCTCP (15.1 μ s vs. 23.4/24.1 μ s). This difference is primarily due to Homa’s use of polling (4 μ s) and its optimization of SoftIRQ core selection (3–4 μ s). In addition, Homa’s single-socket-per-thread approach eliminated

Name	Mean	Description
W2	433	Search application at Google [33].
W3	2423	Aggregated workload from all applications running in a Google datacenter [33].
W4	60175	Hadoop cluster at Facebook [32].
W5	385315	Web search workload used for DCTCP [1].

Table 3: The message size distributions used for benchmarks. The workloads were taken from [25], except that messages larger than 1 MB in W4 and W5 were truncated to 1 MB (the largest size permitted under Homa/Linux). The workloads are ordered by average message size (“Mean”): W2 is most skewed towards small messages and W5 is most heavy-tailed.

the need for `epoll` system calls, which saves 1 μ s. Section 6 shows that latencies in real applications are considerably higher than this for both Homa/Linux and TCP. For comparison, the lowest achievable round-trip latency for this hardware (using kernel bypass) is 3.7 μ s [19].

Homa’s throughput is only about half that of TCP or DCTCP when a single client sends large RPCs back-to-back to an unloaded server (10 Gbps vs. 20 Gbps). This is because Homa’s message-based interface reduces opportunities for pipelining. In particular, Homa/Linux cannot transfer any part of a message to a waiting application until the entire message has arrived, whereas TCP can overlap network transmission and copying to user space. When multiple RPCs are active simultaneously, Homa’s throughput is equivalent to TCP’s, while DCTCP’s throughput is slightly lower.

Table 2 also shows the maximum small message request rate. Homa’s throughput is 60% above TCP or DCTCP. One reason is Homa’s ability to batch received packets more effectively than TCP, as described in Section 4.1. Homa generated average batches of 2–3 packets at the NAPI level, whereas TCP did not batch packets at all for this benchmark.

5.2 Cluster benchmarks

Most of the remaining evaluation is based on a cluster benchmark that uses all 40 nodes. Each node operates simultaneously as both client and server, with multiple client and server threads chosen for each protocol to maximize its performance. Each node has multiple independent Homa server sockets or TCP listen sockets. Clients send request messages to servers chosen at random, and servers return response messages of the same size as the requests. Message sizes are chosen at random to match one of four workload distributions, W2–W5, described in Table 3. These distributions are the same as those used in Montazeri et al. except that W1 is omitted because of space limitations (W1’s behavior is almost identical to W2). The timing of new requests is based on a Poisson arrival function that produces a particular average throughput.

Figure 3 compares Homa with TCP and DCTCP for each of the four workloads under high network loads. The figure uses *slowdown* as a measure of latency, which allows comparisons between messages with different lengths. The slowdown for a given RPC consists of the end-to-end round-trip time (RTT) observed by the client application divided by the RTT for RPCs of the same length measured with Homa under the ideal conditions of Table 2; smaller slowdowns are better.

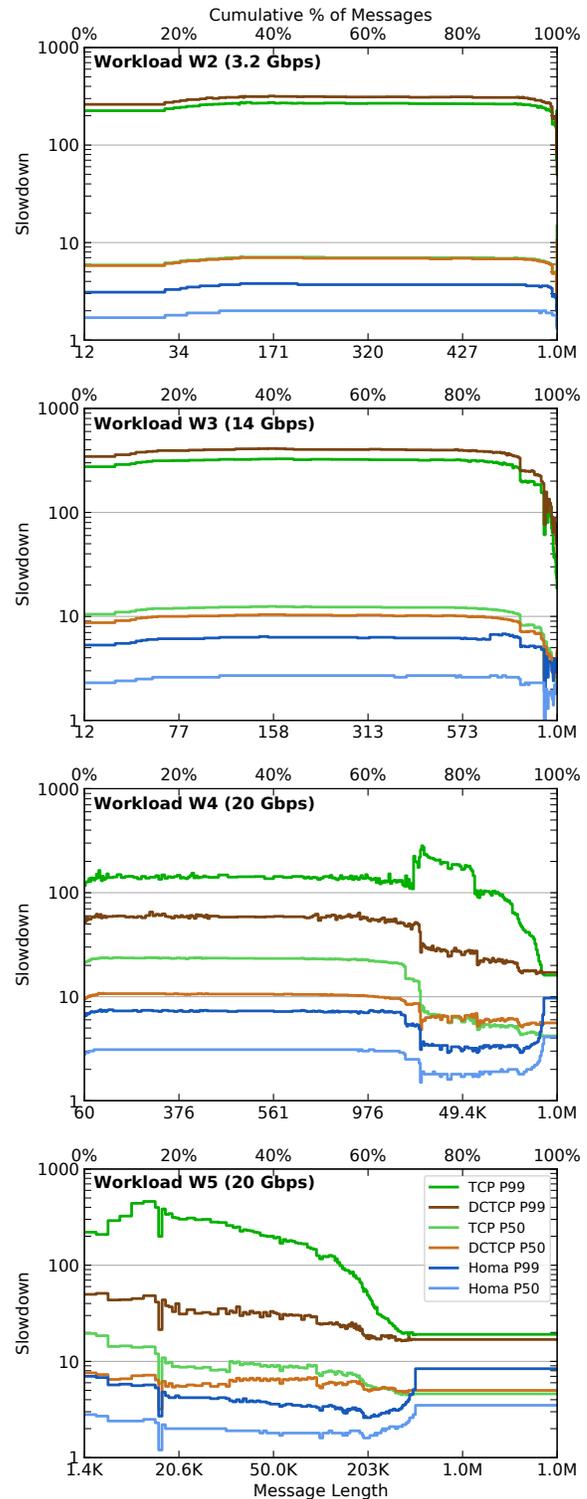


Figure 3: Median and 99th-percentile slowdown as a function of message size for workloads W2–W5. Each x-axis is linear in number of RPCs (i.e. it reflects the CDF of message length for that workload). The network load for each workload (e.g. 20 Gbps for W4 and W5) was chosen so that the protocols operated at 80–90% of their maximum sustainable rates; load is measured in units of message payload bytes sent by each host (an equal amount was also received).

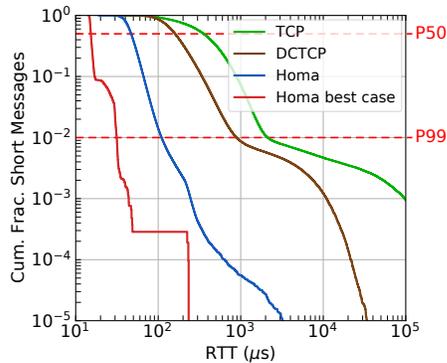


Figure 4: Complementary CDF of round-trip latency for the shortest 10% of RPCs for W4 (messages with 315 bytes or fewer) in Figure 3. A point (x, y) indicates that a fraction y of RPCs had round-trip times longer than x . “Homa best case” was measured under the conditions of Table 2.

Homa’s latencies are significantly lower than both TCP and DCTCP. For example, Homa’s P99 latencies for short messages are 19–72x lower than those for TCP and 7–83x lower than DCTCP. Homa’s P50 latencies for short messages are 3.5–7.5x lower than TCP and 2.7–3.8x lower than DCTCP. Homa’s P99 slowdown is actually lower than P50 slowdown for TCP and DCTCP, except for the the largest messages in W4 and W5. DCTCP generally outperforms TCP, as expected.

W2 is different from the other workloads in that network congestion is not an issue: almost all messages are short, so none of the protocols can support more than about 40% network utilization. For this workload performance is limited primarily by software overheads; Homa still provides much lower latency than either TCP or DCTCP. An analysis of TCP’s P99 behavior showed that it is caused by scheduler anomalies where two threads end up assigned to the same core, even though there are fewer application threads than cores. One of them eventually migrates to a different core, but by the time it resumes execution, many milliseconds have been lost. Anomalies like this did not occur for Homa.

One potential concern with Homa’s SRPT policy is its impact on large messages. Figure 3 shows that this is not a problem in practice. Slowdowns for workloads W4 and W5 do increase for large message sizes, but Homa still beats both TCP and DCTCP on P50 and P99 latency. There is no message size in any workload where TCP or DCTCP outperformed Homa.

Figure 4 shows more detail on the round-trip latency for short messages in W4. Homa’s latencies are better than TCP and DCTCP at every percentile, typically by at least an order of magnitude.

Homa’s slowdown in Figure 4 is entirely due to software overheads: Homa has eliminated network congestion as a significant factor. To verify this, I used a timetracing package to extract precise end-to-end traces of short RPCs with latencies near the 99th percentile. Latency for these RPCs was primarily due to hot spots in load balancing: an application thread cannot execute to handle a short message because its core is

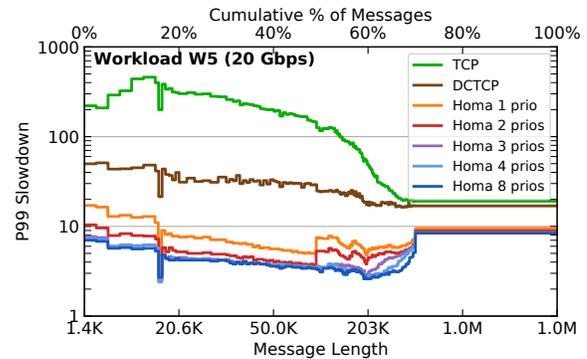


Figure 5: The impact of the number of available priority levels on Homa’s P99 slowdown for workload W5 (other workloads are omitted because of space limitations; they are less sensitive to the number of priorities than W5). TCP and DCTCP are shown for comparison.

occupied by NAPI processing for unrelated large messages. These NAPI bursts can last 100 μ s or more.

In contrast, timetraces for TCP RPCs near P99 showed different causes of tail latency. Roughly two-thirds of the traces had head-of-line blocking where a short message was stuck behind a long one in the same stream. In the remaining traces the extra time occurred after a packet was accepted by the source NIC and before an interrupt occurred on the destination, suggesting buffer buildup in the NIC or at the switch’s egress port. This analysis shows that the P99 latency difference between TCP/DCTCP and Homa is because of protocol features, as opposed to quality of the implementations.

5.3 Number of priorities

One of the concerns that has been raised about Homa is its use of switch priorities (e.g., [21]): priority queues are a limited resource, and in some datacenters they are already allocated to applications with critical QoS needs. Figure 5 shows that Homa/Linux needs only a few levels to achieve maximum performance: it does well with 2 priorities, and additional levels benefit only a small range of message sizes. Even with only a single priority level, Homa’s P99 latency is still much better than DCTCP or TCP.

5.4 Reduced load

The measurements in Section 5.2 were made under high load (80–90% of the maximum that the protocols and network can sustain). Figure 6 shows slowdown for Homa and DCTCP when the load is reduced by a factor of 2x or 10x. Because of space limitations, measurements are shown only for W4, and only for Homa and DCTCP (other workloads are similar).

Figure 6 shows that Homa is more stable than DCTCP. Homa’s latency for short messages varies by less than 2x across a 10x load change, whereas DCTCP’s latency varies by 7x. Homa’s P99 short-message latency at the highest load is 40% lower than DCTCP’s P99 latency at a 10x reduced load.

Homa displays a performance inversion in Figure 6. Once the load drops below 50% of maximum, short-message latency begins to increase; Homa latencies at the lowest load are about 25% *higher* than those at the highest load. The inversion is due to C-states. At low loads, cores may be idle for

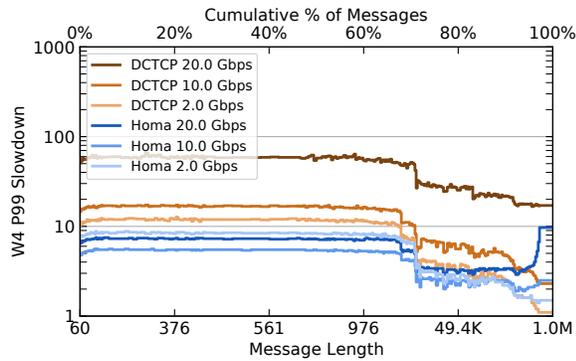


Figure 6: P99 slowdown as a function of message size for W4 under varying loads.

significant periods, causing them to enter C-states; it takes 50–100 μ s to awaken them when packets arrive. I confirmed this behavior by running the benchmark with C-states disabled: in this case, Homa latency continues to improve with lower loads.

Figure 6 suggests that Homa’s automatic priority allocation mechanism may outperform the manual approaches currently used in datacenters. Consider an application using DCTCP whose workload matches W4, and suppose it has exclusive use of the highest network priority. It will then behave roughly as if it had a private network, so its performance will match the DCTCP curve in Figure 6 corresponding to the load generated by that application. Now suppose that, instead of manually allocating QoS levels, all applications switch to Homa, and Homa uses all the priority levels to support them without preference. The performance will now match the Homa curve in Figure 6 corresponding to the combined load of all the applications. There is almost no overlap in these curves: shared Homa provides better latency than differentiated DCTCP for all but the largest 20% of messages. If those particular message sizes are not crucial for the application’s performance, it would be better off in a shared environment under Homa, even though Homa offers no particular QoS support. The shared approach would also eliminate the administrative burden of assigning and managing QoS levels.

5.5 Buffer usage

Homa uses buffers relatively economically compared to TCP, but its performance will degrade if buffer space is exhausted and packets are dropped. Hai et al. measured Homa with switch buffers limited to 200 KB per port [14]. They found that this resulted in very poor Homa performance due to dropped packets and the subsequent timeouts and retransmissions. I analyzed buffer usage in our cluster to get a better understanding of this issue, and found that Homa fits easily in the buffer space available in our 25 Gbps environment.

Hai et al. assumed a fixed-size buffer for each port. However, many modern switches have dynamic buffer pools that can be shared between ports. For example, our switches have 16 MB of buffer space, of which 13 MB can be dynamically shared among 40 ports.

	W2	W3	W4	W5	W5-6K
homa_send	1.35	1.38	0.47	0.45	0.38
homa_recv	2.11	2.54	1.29	1.32	1.22
homa_reply	1.60	1.69	0.56	0.50	0.44
NAPI	1.29	1.76	1.18	1.37	1.01
SoftIRQ	1.31	1.58	0.90	0.87	0.82
Pacer	0.02	0.34	0.71	0.70	0.66
Timer	0.01	0.01	0.01	0.01	0.01
Total	7.69	9.29	5.11	5.03	4.53
Polling	7.30	6.25	1.05	0.18	0.30

Table 4: Total core usage for the components of Homa in the benchmarks from Figure 3; 1.0 corresponds to 100% usage of one hyperthread (work is actually spread across many cores). Polling time is listed separately, and is not included in `homa_recv`. The times for system calls do not include time to enter and leave the kernel. “W5-6K” shows W5 with the MTU increased from 3000 to 6000 bytes.

Using statistics maintained by the switch, I found that Homa’s maximum buffer occupancy across all ports ranged from 246 KB for W2 to 8.5 MB for W5, well within the pool’s 13 MB capacity. To confirm the number for W5, I reduced the pool size to 9 MB and reran the benchmark: its performance was unaffected and the switch recorded no packet drops. I then further reduced the pool size to 7 and then 6 MB. At 7 MB there was still no performance degradation, but Homa dropped 500–1000 packets per second per port under the W5 workload. With 6 MB of buffer space, Homa dropped about 90,000 packets per second per port under W5 (a rate of 0.13%) and performance degraded severely. For comparison, TCP performance dropped noticeably when buffer space was reduced to 10 MB and severely at 6 MB; DCTCP experienced no degradation until buffer space dropped below 2 MB.

Homa’s maximum buffer usage amounts to about 220 KB per port, which is only slightly higher than the value that produced poor results for Hai et al. It seems likely that the dynamic buffer pool explains the difference in our results. Assuming that buffer usage scales with link speed, buffer requirements (and availability) can be characterized in units of Kbytes of buffer space per Gbps of aggregate switch bandwidth. Homa’s maximum usage was 8.5 KB/Gbps, though it performed well with only 7 KB/Gbps. Looking forward to 100 Gbps networks, Broadcom’s Tomahawk 3 switching chip [35] provides 7.5 KB/Gbps of buffer space, which appears to be adequate for Homa. I believe that Homa’s buffer usage can be reduced, but I leave this to future work.

6 Software Overheads

As discussed in the preceding section, Homa/Linux eliminates congestion as a significant performance factor. Tail latency is now limited by software overheads, not network congestion. New congestion control schemes are unlikely to be impactful unless they also reduce software overheads (and simulation results that omit those overheads may be misleading). Significant gains may be had if overheads can be reduced: for example, the RAMCloud implementation of Homa [25] has much lower software overheads than Homa/Linux; as a result, it provides 8x lower P99 latency for short messages in W4

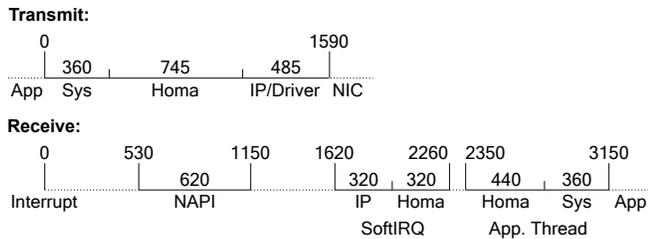


Figure 7: Timelines (in ns) for sending and receiving short messages under the best-case conditions of Table 2; the receiving thread is polling in `homa_recv` when the message arrives (an additional 2000 ns are incurred if the thread is sleeping). All times are in ns. “Sys” refers to Linux code for entering and leaving system calls.

(unfortunately, the RAMCloud implementation is impractical, as discussed below). This section provides more data on software overheads.

As discussed previously, software overheads (in particular, hot spots due to conflicts between the thread scheduler and network load balancers) are the primary source of tail latency for Homa/Linux. In addition, software overheads limit throughput and place excessive demands on cores for packet processing. They particularly impact throughput for short messages: the 1.6 Mops/sec maximum request rate reported for servers in Table 2 consumes resources equal to 18 of the machine’s 20 cores, even though no meaningful work is done in servicing the requests. Table 4 shows core utilization for each of the four workloads; W2 and W3, which have the smallest message sizes, consume almost all of the cores, and yet they are unable to fully utilize a 25 Gbps network. W4 and W5 process larger packets, so they use fewer cores. The pacer thread consumes about 0.7 core for W4 and W5, which have many large messages. Extrapolating from Table 4, Homa/Linux will require at least 18 cores (excluding polling) to drive a 100 Gbps network at 80% utilization.

One possible way to reduce software overhead is to increase the MTU. However, Table 4 shows only small benefits: doubling the MTU from 3000 to 6000 bytes reduces core utilization for W5 only 11%, and workloads with smaller messages benefit even less.

Figure 7 breaks down the best-case latency for sending and receiving short messages. The total round-trip software overhead is about 9.5 μ s, consisting of 1.6 μ s on each node to send a message and about 3.2 μ s on each node to receive one. This is notably higher than the network time (from NIC queue to interrupt), which is about 5.6 μ s per round-trip. Figure 7 and Table 4 show that software overheads are distributed across many components; there is no single culprit to optimize.

Unfortunately, Figure 7 significantly understates the latency for real workloads. Software overheads increase by 2–3x when load balancing is used. This can be seen in the upper left corner of Figure 4. The “Homa best case” RTT is about 15 μ s; in this measurement, all protocol processing occurs on a single core on each machine. However, in the W4 workload, where protocol processing is distributed across cores, the median RTT for short requests is 3x higher (about 47 μ s); only 1% of requests complete in under 30 μ s. An analysis of de-

tailed timetraces showed that each stage of processing from Figure 7 slows down by a factor of 2–3x (presumably due to cache coherency traffic?); again, there is no single culprit.

7 The Future of Transport Protocols

The preceding sections showed that transport protocols implemented in kernel software carry a painfully high cost in terms of both application latency and core usage. Although some improvements may be possible, such as better thread schedulers that reduce hot spots from load balancing, fundamental challenges will remain. This section discusses the possibility of moving transport protocols out of the kernel.

7.1 Moving transport protocols to user space

Several recent projects have explored moving protocol processing to user space, including MICA [22], RAMCloud [29], IX [5], ZygOS [31], Shenango [27], eRPC [19], and Snap [23]. At first glance, these systems appear to reduce software overheads significantly. For example, Shenango and RAMCloud can serve about 1 M requests/sec/core and eRPC can serve 10 M requests/sec/core, vs. just 0.1 M requests/sec/core for Homa/Linux. eRPC offers best-case latency of 3.7 μ s, vs. 15.1 μ s for Homa/Linux, and the RAMCloud implementation of Homa provides P99 latency for W4 less than 15 μ s, vs. about 100 μ s for Homa/Linux.

However, most of these systems have significant simplifications and/or restrictions, such as the following:

- They measure under unrealistic conditions like those in Table 2.
- They don’t do load balancing, or do it in a hand-optimized fashion that eliminates hot spots and the 2–3x load-balancing overhead discussed above.
- The measurement workloads contain only short or only long messages (combined workloads are both more realistic and more challenging).
- They don’t consider congestion control.
- They assume that every application can implement the protocol independently (see below).

Many of the overheads experienced by Homa/Linux cannot be eliminated; for example, at least one core will still be consumed polling the NIC and receiving packets, and if output rate limiting is needed, another core will be consumed for that.

Of the user-space systems listed above, only Snap provides a full-featured production implementation. However, it reduces software overheads by only about 2x compared to Homa/Linux (Snap requires 7–14 cores to drive a 100 Gbps network at 80% load bidirectional, vs. 18 for Homa/Linux; this is still a steep price to pay). Snap appears to incur load balancing overheads similar to those for Homa/Linux. A single Snap core without load balancing can drive a 100 Gbps network at 80% load in one direction, but when load balancing is enabled (as in the numbers for 7–14 cores above) throughput/core drops by 3.5–7x. Furthermore, Snap reported P99 latencies for short messages of 300–400 μ s under high network loads, which is 3–4x higher than Homa/Linux. This suggests

that user-space implementations will not be a panacea for the problem of high software overheads.

One of the challenges with user-space protocol implementations is that they perform best when every application can implement the protocol independently. However, not all protocols fit this model. For example, Homa requires global state for congestion control (for grant management and the pacer). If global state is required, then the protocol must be implemented in a shared service and packets must pass through that service on their way to and from the network. This introduces extra core/thread crossings, adding significant overhead. Snap uses this approach, which could explain why its overhead is higher than many other user-level protocol implementations.

7.2 Moving transport protocols to the NIC

The challenges with software transport protocols will only get worse in the future, since network speeds are increasing while CPU speeds are static. We are approaching a point where it no longer makes sense to implement transport protocols in software. The alternative is to move them entirely to the NIC; applications would communicate directly with the NIC using kernel bypass. Packets would no longer be visible to host software: all communication with the NIC would be in terms of messages. In addition, the NIC would implement intelligent load balancing, such as distributing incoming requests across available threads in a service, so that packets do not need to pass through an additional core just for load balancing.

Moving the transport protocol to the NIC would reduce end-to-end application latency by at least 5x, increase small-message throughput by 5–10x, and use silicon real estate more efficiently, freeing cores currently used for inefficient protocol processing so they can run application code instead. Reducing end-to-end latency would also reduce RTTbytes, cutting buffer consumption in the switches. This transition could be as impactful for system performance as the transition from programmed I/O to direct memory access in the 1960's.

Designing such a NIC will be challenging; it will require a new architecture that combines a line-rate packet processing pipeline with enough programmability to allow a variety of transport features and to support easy transport maintenance and evolution. The NIC will also need to support network virtualization features commonly implemented in software today [30, 9]. There exist systems that provide some of the required features, but none is fully satisfactory. For example:

- RDMA NICs provide kernel bypass and low latency, but their mechanisms for congestion control and load balancing are inadequate. In addition, existing NICs are not open or programmable.
- Today's "smart NICs" have inadequate performance and/or programmability. Smart NICs come in two flavors. The first is implemented with many general-purpose cores. These NICs still implement transport protocols in software, with all the performance problems discussed above. The second flavor is based on FPGAs; they can potentially provide adequate performance, but are difficult to program.

- New packet processing pipelines such as P4 [6, 7] operate at line rate and are programmable, but they do not currently have enough power to implement transport protocols. As one example, P4 does not provide long-lived state that is required for transport protocols.

One promising direction of research is to extend P4 with additional mechanisms to meet the needs of transport protocols [18]; this work is still embryonic.

7.3 The future of TCP

TCP has a long and illustrious history, and it has found use across an extraordinary range of technologies and environments. However, datacenters did not exist when TCP was designed, and virtually every aspect of the TCP design is wrong for the datacenter:

- Its connections create high space and time overheads.
- Its stream orientation is awkward for applications and causes high tail latency due to head-of-line blocking.
- Its fair scheduling increases tail latency for all message sizes.
- Its sender-driven congestion control ensures buffer occupancy at high loads, which drives up tail latency.
- It doesn't take advantage of in-network priority queues.
- It requires in-order delivery, restricting opportunities for load balancing both in network hardware and host software.

Furthermore, TCP's high implementation complexity will make it difficult to implement in hardware. Thus, it is difficult to imagine a path to high performance datacenter networking that is based on TCP.

8 Related Work

The last decade has seen many proposals to improve TCP performance and/or solve the congestion control problem for datacenters. Examples include DCTCP [1], D³ [37], HULL [2], D²TCP [36], PDQ [17], pFabric [3], PIAS [4], QJUMP [13], pHost [11], Karuna [8], and NDP [15]. Of these, DCTCP appears to be the only one with a readily-available Linux implementation that can be compared with Homa/Linux.

Homa's congestion control mechanism is effective against congestion at the network edge, but it does not address congestion in the core. Several other recent projects have addressed core congestion, including TIMELY [24], HPCC [21], and Swift [20]. Swift and Homa could probably be synergistically combined, with Swift dynamically adjusting RTTbytes to manage congestion in the core while Homa eliminates it at the edge. At the same time, it seems possible that core congestion is due primarily to TCP's requirement of flow-consistent routing, and that a datacenter using Homa with packet spraying would not experience significant core congestion; this would be an interesting experiment for future work.

Aeolus [14] proposed modifications to Homa to improve performance when buffer space is exhausted. However, measurements in Section 5.5 indicate that buffer exhaustion is unlikely to occur in modern switches with shared buffer pools, so the Aeolus modifications are not necessary.

As mentioned in Section 6, several recent projects have developed network transports outside the kernel in order to avoid the overheads of an in-kernel transport. Examples include MICA [22], RAMCloud [29], IX [5], ZygOS [31], Shenango [26, 27], eRPC [19], and Snap [23].

9 Conclusion

This paper has demonstrated two things. First, it has shown that the Homa transport protocol can be implemented in a practical setting that allows it to be used by a variety of applications. Homa/Linux retains the benefits of Homa's congestion-control mechanism and outperforms TCP and DCTCP by a wide margin, offering order-of-magnitude lower tail latencies across a range of workloads and message sizes.

Second, this paper has shown that the battle for networking performance is shifting from network protocols to software overheads. Increasing network speeds make it ever more difficult to do protocol processing in software, and overheads increase as more cores are harnessed. If these overheads can be eliminated, for example with new NIC architectures, another factor of 5–10x in networking performance is possible.

10 Acknowledgments

This work was supported by Cisco, Fujitsu, Google, Huawei, NEC, and VMware. Bare-metal computing resources were provided by CloudLab [10]. Collin Lee, Yilong Li, Amy Ousterhout, and Seo Jin Park provided helpful comments on drafts of this paper. The paper also benefited from comments and suggestions from Michio Honda, the paper's ATC shepherd, and anonymous reviewers, as well as copy editing by Geoff Kuenning.

11 Appendix

This appendix provides additional details on a few aspects of the Homa/Linux implementation.

11.1 GRO packet batching

As discussed in Section 4.1, Homa/Linux collects incoming packets into batches for SoftIRQ processing without regard to message structure. Unfortunately, the Linux infrastructure segregates incoming packets using a hash of packet header fields. When a new packet arrives, Linux uses its hash to find a list of packets being held for batching (if any) that match that hash. Then it passes the new packet and the held list to a transport-specific function. The transport-specific function can batch the packet with an existing packet on the list (by incorporating it into a sublist within the existing packet). If the transport-specific function chooses not to batch the new packet with an existing one, then Linux adds the new packet to the list as a "root" for future batching.

This default mechanism prevents transports from batching packets that hash to different lists. However, the transport-specific function has a third option, which is to indicate that it has completely processed the packet, so Linux should not add it to the list or take any other actions. Homa/Linux takes

advantage of this feature. Homa/Linux keeps track of a distinguished held list for each core (the one corresponding to the first packet received by that core). When packets arrive for any other held list, Homa/Linux ignores that list, merges the packet with the first Homa packet on the distinguished list instead, and indicates to Linux that the packet has been fully processed. As a result, all incoming Homa packets are merged together into a single batch. When the batch is transmitted to SoftIRQ, the distinguished list is reset.

11.2 skbuff management

Homa/Linux uses regular Linux skbuffs for packet buffering. It increments the reference count on transmitted skbuffs in order to retain them until they have been acknowledged. The original skbuff cannot be retransmitted because transmission is not idempotent; for example, lower-level protocol headers get prepended as the buffer traverses the IP stack. Thus, if retransmission is required, Homa/Linux copies the skbuff's data into a new skbuff.

Homa/Linux keeps the skbuffs for an incoming message in a list sorted by offset in the message. New packets are inserted into the list starting at the back (highest offset). With this approach, packet insertion will not require traversing many list elements unless the packet has been delayed a long time.

11.3 Synchronization details

Homa/Linux's approach to socket locking allows it to avoid the awkward mechanism used elsewhere in Linux for locking sockets in SoftIRQ handlers (a SoftIRQ handler could have interrupted a background thread that holds the socket lock needed by SoftIRQ, so SoftIRQ cannot wait for socket locks; when it finds a locked socket, it defers packet processing until the thread releases the lock). Instead, Homa/Linux uses spinlocks for sockets, with interrupts disabled. This prevents an application thread from being interrupted while holding a socket lock.

Homa/Linux takes advantage of the Linux RCU mechanism to prevent sockets from being deleted while operations are underway on them. This eliminates the need to acquire socket locks in some situations, and is particularly useful in situations where it would have been necessary to acquire the socket lock before acquiring RPC locks (this would violate ordering constraints necessary to prevent deadlock).

References

- [1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.
- [2] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 19–19, Berkeley, CA, USA, 2012. USENIX Association.

- [3] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM 2013 Conference, SIGCOMM '13*, pages 435–446, New York, NY, USA, 2013. ACM.
- [4] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. Information-agnostic Flow Scheduling for Commodity Data Centers. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, NSDI'15*, pages 455–468, Berkeley, CA, USA, 2015. USENIX Association.
- [5] 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, Broomfield, CO, Oct. 2014. USENIX Association.
- [6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87—95, July 2014.
- [7] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, SIGCOMM '13*, page 99–110, New York, NY, USA, 2013. Association for Computing Machinery.
- [8] L. Chen, K. Chen, W. Bai, and M. Alizadeh. Scheduling Mixflows in Commodity Datacenters with Karuna. In *Proceedings of the ACM SIGCOMM 2016 Conference, SIGCOMM '16*, pages 174–187, New York, NY, USA, 2016. ACM.
- [9] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermano, E. Rubow, J. A. Doucuer, J. Alpert, J. Ai, J. Olson, K. DeCabooteer, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387, Renton, WA, Apr. 2018. USENIX Association.
- [10] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.
- [11] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. pHost: Distributed Near-optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT '15*, pages 1:1–1:12, New York, NY, USA, 2015. ACM.
- [12] Google. gRPC: A High Performance, Open-Source Universal RPC Framework. <http://www.grpc.io>.
- [13] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. 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, Oakland, CA, 2015. USENIX Association.
- [14] S. Hai, W. Bai, G. Zeng, Z. Wang, B. Qiao, K. Chen, K. Tan, and Y. Wang. Aeolus: A Building Block for Proactive Transport in Datacenters. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM '20*, page 422–434, New York, NY, USA, 2020. Association for Computing Machinery.
- [15] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichik, and M. Mojsik. Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the ACM SIGCOMM 2017 Conference, SIGCOMM '17*, pages 29–42, New York, NY, USA, 2017. ACM.
- [16] Homa GitHub repository. <https://github.com/PlatformLab/HomaModule>.
- [17] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *Proceedings of the ACM SIGCOMM 2012 Conference, SIGCOMM '12*, pages 127–138, New York, NY, USA, 2012. ACM.
- [18] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, C. Kim, and N. McKeown. The nanoPU: Redesigning the CPU-Network Interface to Minimize RPC Tail Latency. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021.
- [19] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, Feb. 2019. USENIX Association.
- [20] G. Kumar, N. Dukkupati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, D. Wetherall, and A. Vahdat. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM '20*, pages 514–528, New York, NY, USA, 2020. Association for Computing Machinery.
- [21] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu. HPC: High Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM '19*, pages 44–58, New York, NY, USA, 2019. Association for Computing Machinery.
- [22] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, Apr. 2014. USENIX Association.
- [23] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkupati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Mutschick, 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, 2019. Association for Computing Machinery.
- [24] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 537–550, New York, NY, USA, 2015. ACM.
- [25] B. Montazeri, Y. Li, M. Alizadeh, and J. 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*, SIGCOMM '18, pages 221—235, New York, NY, USA, 2018. Association for Computing Machinery.
- [26] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, Feb. 2019. USENIX Association.
- [27] A. E. Ousterhout. *Achieving High CPU Efficiency and Low Tail Latency in Datacenters*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2019.
- [28] J. Ousterhout. *A Philosophy of Software Design*. Yaknyam Press, Palo Alto CA, USA, 2018.
- [29] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, et al. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS)*, 33(3):7, 2015.
- [30] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, May 2015. USENIX Association.
- [31] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.
- [32] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, SIGCOMM '15, pages 123–137, New York, NY, USA, 2015. ACM.
- [33] R. Sivaram. Some Measured Google Flow Sizes (2008). Google internal memo, available on request.
- [34] Apache Thrift. <https://thrift.apache.org>.
- [35] BCM56960 Series: High-Density 25/100 Gigabit Ethernet StrataXGS Tomahawk Ethernet Switch Series. <https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56960-series>.
- [36] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware Datacenter TCP (D2TCP). In *Proceedings of the ACM SIGCOMM 2012 Conference*, SIGCOMM '12, pages 115–126, New York, NY, USA, 2012. ACM.
- [37] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 50–61, New York, NY, USA, 2011. ACM.

Ayudante: A Deep Reinforcement Learning Approach to Assist Persistent Memory Programming

Hanxian Huang Zixuan Wang Juno Kim Steven Swanson Jishen Zhao
University of California, San Diego

Abstract

Nonvolatile random-access memories (NVRAMs) are envisioned as a new tier of memory in future server systems. They enable a promising persistent memory (PM) technique, with comparable performance of DRAM and the persistence property of storage. However, programming PM imposes non-trivial labor effort on writing code to adopt new PM-aware libraries and APIs. In addition, non-expert PM code can be error-prone. In order to ease the burden of PM programmers, we propose *Ayudante*¹, a deep reinforcement learning (RL)-based PM programming assistant framework consisting of two key components: a deep RL-based PM code generator and a code refining pipeline. Given a piece of C, C++, or Java source code developed for conventional volatile memory systems, our code generator automatically generates the corresponding PM code and checks its data persistence. The code refining pipeline parses the generated code to provide a report for further program testing and performance optimization. Our evaluation on an Intel server equipped with Optane DC PM demonstrates that both microbenchmark programs and a key-value store application generated by *Ayudante* pass PMDK checkers. Performance evaluation on the microbenchmarks shows that the generated code achieves comparable speedup and memory access performance as PMDK code examples.

1 Introduction

Enabled by nonvolatile random-access memory (NVRAM) technologies, such as Intel Optane DIMM [33], persistent memory (PM) offers storage-like data persistence through a fast load/store memory interface [5, 63]. PM is envisioned to be a new tier of memory in future servers with promising benefits, such as fast persistent data access and large capacity.

However, the PM technique also introduces substantial challenges on programming. First, currently the burden of PM programming is placed on system and application programmers: they need to implement new PM programs or rewrite

legacy code using PM programming libraries with a variety of programming interfaces and APIs [10, 16, 17, 22, 27, 31, 73]. Despite the promising development of libraries, implementing PM programs requires non-trivial labor efforts on adopting new libraries, debugging, and performance tuning. Second, the storage-inherited crash consistency property demands rigorous data consistency of the stand-alone memory system [44, 54] to recover data across system crashes; fully exploiting NVRAM's raw memory performance [15, 34, 76, 77, 80] requires programs to minimize the performance overhead of crash consistency mechanisms. As such, programmers need to understand the durability specifications and ordering guarantees of each PM library in order to provide sufficient persistence guarantee, while avoiding the performance overhead of adding unnecessary persistence mechanisms. Third, because various PM libraries provide different programming semantics, programmers need to manually transform the semantics, when switching from one PM library to another. As a result, PM programming is a tedious and time-consuming task even for the expert PM programmers, while imposing a painful learning period for non-experts. Moreover, the challenges significantly prolong the software development process and hold back the wide adoption of the PM technique.

In order to address these challenges, we propose *Ayudante*, a deep reinforcement learning (RL)-based PM programming framework to assist PM programming by transforming volatile memory-based code into corresponding PM code with minimal programmer interference. *Ayudante* consists of two key components as illustrated in Figure 1. First, *Ayudante* employs a deep RL-based code generator to automatically translate volatile memory-based C, C++, or Java code into corresponding PM code, by inserting PM library functions and instructions. Second, we design a code refining pipeline to parse the generated code to provide a report for programmers to further test, debug, and tune the performance of the code after the inference of our RL model. *Ayudante* intends to save the time and energy of programmers on implementing PM code from scratch. As such, *Ayudante* allows programmers to focus on leveraging or implementing volatile

¹Ayudante source code: [1]

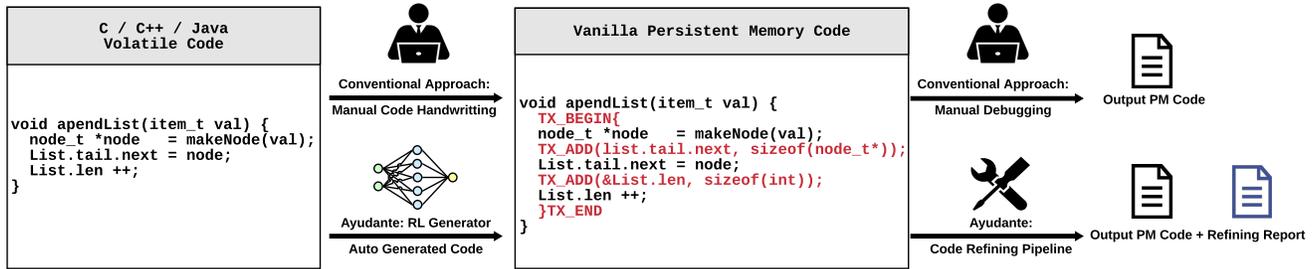


Figure 1: Ayudante framework overview. The framework takes conventional C, C++, or Java source code for volatile memory as an input and generates the corresponding PM code. Ayudante leverages a RL-based method to automatically translate the volatile version of source code to a nonvolatile version by inserting proper PM library annotations. In addition, Ayudante generates a code refining report to help reduce bugs and improve run-time performance.

memory-based code on traditional programming semantics that they are familiar with.

Due to the similarity to neural machine translation (NMT) problems in natural language processing (NLP), program translation is recently demonstrated to have promising results by adapting NMT techniques, such as sequence-to-sequence models [35, 79], word embedding [9, 12, 56], and tree-to-tree LSTM encoder-decoder on abstract syntax tree (AST) [11, 14]. However, the existing machine learning (ML)-based program translation methods focused on simple programs and data structures; the models fall short of handling sophisticated program syntax, data structures, and consistency reasoning, which yield large and complex search spaces [11, 14]. To this challenge, we integrate our RL model with Monte-Carlo tree search and carefully design our neural network architectures to improve generation efficiency. Furthermore, we adopt transfer learning to train the model for Java code generation based on the model trained for C and C++ languages to reduce training time. In summary, this paper makes the following contributions:

- We propose *Ayudante*, the first deep RL-based PM programming assistant framework, which automatically transforms volatile memory code to PM code. Our RL model mimics the behavior of expert PM programmers navigating through the input source code to add proper PM functions and instructions. We augment the RL model with Monte-Carlo tree search strategy to achieve efficient generation.
- We leverage a novel transfer learning model to transfer the PM programming semantics from the existing libraries of one programming language to another. In particular, this paper shows an example of transferring the knowledge of PM programming semantics from C/C++ to Java, saving training time for Java-based PM code generator. This approach sheds light on adapting PM code generation in various languages at low extra effort.
- We evaluate Ayudante with microbenchmarks incorporating various data structures and a key-value store application. Our results show that all the generated PM code passes PMDK checkers, with comparable performance on an Intel

Optane DC PM server as code handwritten by experts.

- Ayudante assists novice PM programmers by reducing their time and energy spent on learning new PM libraries, automating the modifications on legacy code, and facilitating bug detection and performance tuning.

2 Background and Motivation

We motivate our Ayudante framework by PM programming challenges and opportunities in deep RL.

2.1 PM Programming Common Practice

PM systems introduce a new set of programming semantics that diverges from the conventional storage systems programming. Instead of extensively relying on slow system calls to access the persistent data, programmers now directly communicate with the byte-addressable nonvolatile main memory (NVMM) using load and store instructions. As PM combines the traits of both memory and storage, PM system requires crash consistency without hurting the memory-like access performance. One common practice of PM programming is to first use a PM-aware filesystem [20, 78] to manage a large memory region in NVMM. An application can then use a direct access (DAX) `mmap()` system call provided by the filesystem to map a nonvolatile memory region into its address space. From there, the application can directly access the NVMM. This programming model is portable and achieves high performance by reducing costly system calls that are on the critical paths [34, 80].

The PM programming model avoids directly using filesystem system calls for data accesses, making it difficult to use the conventional storage system’s crash consistency and failure recovery mechanisms that extensively use system calls. As a result, PM programs need to maintain crash consistency and develop recovery mechanisms by themselves, rather than simply relying on the underlying filesystems. It is the programmers’ responsibility to provide the crash consistency along with a proper recovery mechanism. Because PM programs rely on load and store instructions to access PM, a single mis-ordered store instruction or a missing cacheline

flush and write-back may make it impossible to recover to a consistent state after a system crash. As such, PM programs need to adopt (i) cacheline flush or write back instructions to ensure that data arrives at NVMM [20] and (ii) proper memory fence instructions to maintain the correct memory ordering [20]. To simplify the crash consistency guarantee in programming, many PM libraries [17, 22, 74] support fine-grained logging of the data structure updates, checkpointing [78], shadow paging [32], or checksum [40, 62] as the most common approaches to enforce failure atomicity to guarantee the recovery of data.

2.2 Challenges of PM Programming

Despite the promising performance and portability, the PM programming model imposes substantial challenges to software programmers on ensuring crash consistency and failure recovery, debugging, and performance optimization.

Labor Efforts of Adopting PM Libraries. Each PM library, such as PMDK [17], Atlas [10], go-pmem [22], and Corundum [31], typically defines its own failure model and provides a set of new programming semantics to hide the instruction level details. Although the high-level programming semantics ease the programming burden, it is still laborious and error-prone to use those libraries as shown in Figure 3. In order for programmers to correctly use those libraries, they must learn the APIs and integrate them into the legacy code after fully understanding their own programming models.

We investigate code modification efforts of various PM programs as shown in Figure 2. It requires the change of 1547 lines of code (LOC) in order to transfer a volatile version of Memcached [28] into PM version [48] using PMDK library [17], which is 14% of the Memcached source code. Figure 2 also shows that other widely used data structures supported by PMDK require at least 15% LOC changes. It is invasive to perform such intensive modifications to a code base that is already complex and highly optimized for volatile memory operations.

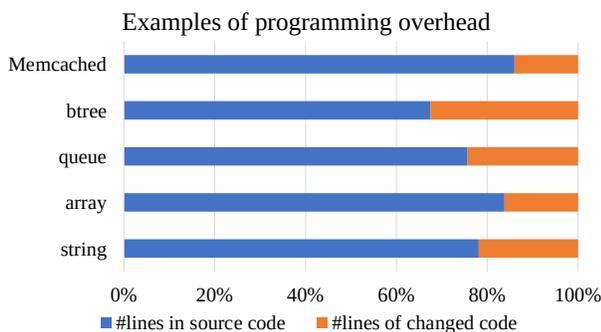


Figure 2: Proportion of lines of changed code to adopt a PM library, with PMDK microbenchmarks and a Memcached application.

Error-prone Non-expert Code and Debugging. Despite the high-level programming semantics provided by PM libraries,

writing a PM code can be error prone as shown by previous studies [44, 45, 54]. It is typically programmer’s responsibility to test and debug PM programs. Many crash consistency related bugs in PM code are hard to identify, as they may not interfere with program execution until the recovery stage after a system crash. Figure 3 shows such an example, where a programmer forgets to add a snapshot API offered by a PM library. The example executes normally but the program recovers to an undefined, non-consistent state undetected. Such bugs are not intuitive, and therefore extremely difficult to debug. Recent development of PM debugging tools leads to promising results [44, 45, 54]. However, the tools still rely on programmer’s experience and knowledge on PM to annotate or navigate through the programs find bugs, making it hard to use by non-expert programmers.

```

1 int Queue::enqueue(...) {
2     ...
3     TX_BEGIN(pop) {
4         TX_ADD_DIRECT(&queue->back);
5         queue->back += 1;
6         TOID(struct entry) entry =
7             TX_ALLOC(struct entry, sizeof(struct entry) + len);
8         D_RW(entry)->len = len;
9         memcpy(D_RW(entry)->data, data, len);
10        // the following snapshot code is missing:
11        // TX_ADD_DIRECT(&queue->entries[pos]);
12        queue->entries[pos] = entry;
13    } TX_ONABORT {
14        ret = -1;
15    } TX_END
16 }

```

Figure 3: A buggy example of enqueue implementation using PMDK. The programmer forgets to add a snapshot function, and therefore violates the crash consistency requirement.

Performance Tuning. PM programming requires the cacheline flushes and fences to ensure that data updates arrive at NVMM in a consistent manner [13, 85]. These instructions introduce performance degradation by defeating the processor’s hardware performance optimization mechanisms, including caching and out-of-order execution. Although it is generally acceptable to sacrifice certain levels of run-time performance to maintain data persistence, unnecessary cacheline flushes and fences will significantly hurt system performance, rendering undesirable deployment performance of PM in production environments. Therefore, it is essential to avoid using unnecessary cacheline flushes and fences. Furthermore, various PM libraries relax the crash consistency guarantee at different degrees for performance optimization. It is the programmers’ responsibility to cherry-pick the optimal library for their application which in turn requires a vast amount of experience and prior knowledge.

2.3 Deep RL for PM Programming

We identify deep RL as a promising technique to perform automatic PM code generation, due to its powerful learning capability in tasks with a limited amount of available training

data [21, 29]. RL [69, 70] is a category of ML techniques that tackles the decision making problems. Deep RL [21, 29, 50, 51] augments deep neural networks with RL to enable automatic feature engineering and end-to-end learning [21]. We observed that deep RL is a better solution than RL for problems with higher dimensional complexity as it does not rely on domain knowledge. Due to these promising features, deep RL is widely used in games [51, 53, 68], robotics [36, 38, 60], and natural language processing [65, 75].

No end-to-end frameworks currently exist to automatically translate a piece of volatile memory code into PM code and perform crash consistency checking. State-of-the-art PM libraries [17, 22, 27, 31, 42, 47] and debugging tools [44, 45, 54] require programmers to annotate the code. Jaaru [24] does not require programmer’s annotation but it relies on the program crash to detect bugs. It is also impractical to enumerate all the possible changes of each line of the code, while passing the checks of compilers and debugging tools. To address these issues, it is critical to automate PM code transformation, while achieving a high PM checker passing rate at a low transformation cost.

Fortunately, translating a volatile memory code into PM code can be formulated as a decision problem for sequential code editing, which is considered as solvable by deep RL. Moreover, augmented with Monte-Carlo tree search [8, 18] – a type of look-ahead search for decision making – deep RL is able to search the decision results more efficiently and effectively. Previous automatic code generation and translation studies focus on translation between different programming languages [41] and addressing debugging syntax errors [26]. To our knowledge, this paper is the first to formulate a ML-based PM program translation problem.

3 Ayudante Design

To embrace the opportunities and address the challenges described above, we propose Ayudante, a deep RL-based programming assistant framework as illustrated in Figure 1.

3.1 Ayudante Framework Overview

Ayudante consists of two key components: a deep RL-based code generator and a code refining pipeline. Our code generator takes conventional source code developed for volatile memory systems as the input and generates a vanilla PM code through a RL model. We design our RL model to mimic programmer’s behavior on inserting PMDK library annotations into the volatile version of code. In order to reduce the training time and effort, we first train our model for C/C++ code by RL, and then employ transfer learning to adapt our model to Java programs. We show that our model is generalizable to various PM programs in our test set on the open source Leetcode solution programs (Section 5). Our code refining pipeline integrates multiple PM checking and debugging tools to generate a report of syntax bugs (if any) and suggestions on run-time performance optimization. The report allows programmers to further improve and test the

code.

Ayudante offers the following promising automated characteristics in assisting PM programming:

- **Efficient PM code generation** through a deep RL model augmented with Monte-Carlo tree search, which efficiently searches the correct code edits with significantly smaller search space.
- **Reduced bugs** through a deep RL model pre-trained to avoid bugs detected by checkers in the training environment.
- **Code refining reports and improved performance** through a code refining pipeline for programmers to further inspect the possible improvements to the generated programs if necessary.

3.2 Deep RL-based Code Generator

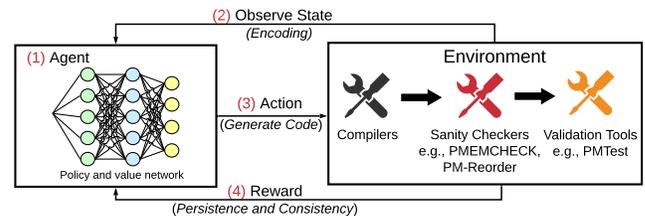


Figure 4: Ayudante’s deep RL model consists of an agent and an environment. During model training, the agent repeatedly generates and sends actions to the environment based on the rewards and states it receives.

We design a deep RL network that generates the PM code. The trained network mimics the programmer’s behavior; it navigates through the input source programs and adds proper code in the corresponding locations. Figure 4 shows Ayudante’s deep RL model, which consists of (a) an agent with a *policy and value network* and (b) an *environment*. The policy and value network responds to a state (the encoded input source code) and outputs an action (which navigates through source code and inserts PM library functions). The environment applies the action to the last code state to generate a new code state, and tests it by a set of PM code checkers to return a reward to the agent. Details of integrating various PM checkers in the environment are discussed in Section 4.4.

The model is trained on a volatile version of PMDK example code [17] (by removing PM annotations). During training, the policy and value network is updated for a better action policy according to the reward function. After training offline, the RL generator performs online inference to generate the best actions and output a PM code according to the pre-trained policy and value network. We test on the data structures from open-source Leetcode solution code [6, 61, 83]. In the following, we describe the details of our RL and transfer learning models. Section 4 will discuss detailed implementation of training and inference of our models and the datasets used for training and testing.

3.2.1 Agent, State, Action, and Reward

As shown in Figure 4, our RL model consists of four major components: agent, state, action, and reward.

Agent. An agent is an executor of actions on a state. In our model, a state is a representation of source code, while an action is navigating through source code or inserting one PM library API function to source code. The agent repeatedly obtains a new state from the environment, executes actions on the state, and re-compiles the modified code if necessary, until the model outputs a complete PM code that passes the checkers in the environment.

State. A state is a pair of *string* and *position*. A string is a source code in plain text, while a position represents the current position in the code to take an action at. To preprocess the input and make the source code recognizable by the RL model, we perform an *encoding* operation, which transforms the source code into a sequence of *tokens* to feed into the RL model. The encoding using an Encoder-Decoder long short-term memory (LSTM) autoencoder [30] to fit sequence data. Once the autoencoder fits, the encoder in the model is used to encode or compress sequence data as a feature vector input to the RL model. Each encoded token represents either the program text or a program location for an API insertion. The string is initialized in a tokenized form of the input source code. The program location token is initialized to the first token in the program. We encode the program text, augmented with the program location using an LSTM network [30] and feed the state into the policy and value network. When taking an action, the modification is executed on the original program by the environment before compilation and testing.

Action. Our RL model has two types of actions: (1) navigation, which jumps to a new program location and (2) edit, which modifies the string (i.e. modifies the source code). The two actions are non-overlapping: navigation action does not update the code, while edit action only adds a PM library API function to the current program location without changing the location. We define two navigation actions, move-down and move-out. Move-down sets the program location to the next token. Move-out moves the program location to the current curly braces. A combination of these two navigation actions allows the model to enumerate all possible insertion locations of PM library APIs.

The edit action utilizes the possible library API annotations, which we manually select from PMDK [17]. The edit action either annotates one line of code or wraps a block of source code into a pair of annotations (e.g., wrapping with `TX_BEGIN` and `TX_END`). We do not consider deletion of previous API annotations, because it is identical to do nothing in the deleted locations. The invalid edit action that causes syntax errors or bugs will be rejected by the environment and punished by the reward function.

Reward. In a typical RL model, a reward is used to measure the consequences of the actions and feedback to the agent

to generate better actions. In our training process, the agent performs a set of navigation and edit actions to the program, and receives either the maximum reward if the generated code passed all the PM checkers in the environment, or a small reward formulated as follows, which consists of three penalties – step, modification, and code error reported by checkers, respectively:

$$r_t = \phi_1 \cdot \ln S + \phi_2 \cdot \ln M + \sum_{i=1}^n \rho_i \cdot E_i \quad (1)$$

The step penalty is defined by a penalty factor ϕ_1 and a step number S . A step refers to an iteration of taking a new action and feedback the corresponding reward. In each step, the agent is penalized with a very small step penalty (with a small ϕ_1) to motivate the agent to take fewer steps.

The modification penalty is defined by a factor ϕ_2 and a modification number M . ϕ_2 penalizes unnecessary edits and encourages the RL generator to complete the PM code with fewer modifications.

The code error penalty is defined as a summation of penalties given by multiple testing tools. For tool i , ρ_i represents the impact factor of the tool (i.e. how important the tool is), while E_i is the number of errors reported by the tool. The code error penalty penalizes those actions that introduce bugs and encourages the RL model to learn to avoid the bugs detectable by the checkers in the environment. The ρ_i can be tuned to give more attention to testing tool i , so as to reduce the corresponding bugs detectable by this tool in the ultimately generated code. In our model, the number of errors E_i is a few magnitudes lower than the number of steps S . Therefore, we use $\ln S$ in the reward instead of S to balance the penalties. The same reason applies to $\ln M$.

3.2.2 An Example of RL-based Code Generation

Figure 5 shows an example PM code generated by our RL model. To generate such code in inference, our trained model takes an input of a pure C code (in black color in Figure 5) that is never seen during training. Our model encodes the code into tokens, then performs actions on the tokens as marked by arrows in Figure 5. At each step t , the agent of the model (i) retrieves a state $s_t \in S$ from the environment and (ii) selects an action a_t from all candidate actions with the maximum conditional probability value generated by the policy and value network. In Figure 5, the agent first chooses navigation actions for step ① and ②, then an edit action for step ③. At this point, as the agent changes the state (the source code), the environment executes the edit action to generate a new state s_{t+1} and a corresponding scalar reward r_t . Similarly, the agent performs edit actions at steps ⑦ and ⑩, and therefore generates a complete PM code. The generated code is further verified by the environment using a pipeline of multiple PM testing tools shown in Figure 4. In this example, the generated code passes all the tests. The code refining pipeline

(Section 3.3) will automatically generate code refining suggestions if it identifies bugs or performance degradation that requires programmer’s attention.

```

1  int64_t Queue::pop(){
2  int64_t ret = 0;
3  auto pool = pmem::obj::pool_by_vptr(this);
4  obj::transaction::run(pool, [this, &ret] {
5      if (head == nullptr)
6          throw std::runtime_error("Empty queue");
7      ret = head->value;
8      auto n = head->next;
9      obj::delete_persistent<Node>(head);
10     head = n;
11     if (head == nullptr) tail = nullptr;
12 });
13 return ret;
14 }

```

Figure 5: An example of a sequence of actions taken by the trained agent to generate PM code based on its volatile memory version.

3.2.3 Policy and Value Network

The *policy and value network* (Figure 4) determines which action to take based on an action-value function. In Ayudante, we use deep Q-learning [50, 72], a representative and widely adopted policy and value network. Q-learning is model-free and generalized not to depend on any specific RL models. In Q-learning, the function Q calculates an expected reward given a set of states and actions, which includes the reward of future subsequent actions. In Ayudante, we use deep Q-learning to combine Q-learning with a deep neural network to form a better end-to-end generator. The function Q is defined as:

$$Q(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \cdot \max_a Q(s_{t+1}, a)) \quad (2)$$

where t represents a time step that requires an action to be taken; s_t is a sequence of actions; a_t is the action; r_t is the reward. The decay rate is $0 \leq \alpha \leq 1$. The discount factor is $0 \leq \gamma \leq 1$. We apply such a Q function to our Deep Q Network, working as an iterative decision making algorithm outlined in Algorithm 1. The objective of deep Q-learning is to minimize $(Y - Q(\phi, a; \theta))^2$ based on sequences of actions and observations $s_t = x_1, a_1, x_2, a_2, \dots, a_{t-1}, x_t$. Here, Y represents the expected reward, while $Q(\phi; a; \theta)$ is the reward calculated from the Q function, with trainable weight parameters θ . The Q function works on fixed-length representation of code modification histories collected by function ϕ . M is the maximum number of epochs and T is the number of iteration to modify the code used to simulate the descent process, which are user-defined parameters. This process will also generate the training datasets of Q-learning, stored in a replay memory \mathcal{D} , with a maximum capacity N . When \mathcal{D} is inquired for a minibatch of transitions, it will return Y_j and $Q(\phi, a_j; \theta)$.

Algorithm 1 The policy and value network function.

- 1: Initialize replay memory \mathcal{D} to capacity N and random initialize θ
- 2: **for** epoch from 1 to M **do**
- 3: Initialize sequence $s_1 = \{x_1\}$ and $\phi_1 = \phi(s_1)$
- 4: **for** t from 1 to T **do**
- 5: With probability $< \epsilon$ select a random action a_t
- 6: Otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
- 7: Execute action a_t : navigate or insert an API
- 8: Get reward r_t , and next state x_{t+1}
- 9: Set $s_{t+1} = s_t, a_t, x_{t+1}$ and $\phi_{t+1} = \phi(s_{t+1})$
- 10: Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}
- 11: Sample a minibatch of transitions $(\phi_t, a_t, r_t, \phi_{t+1})$ from \mathcal{D}
- 12: Set $Y_j = r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta)$ for a non-terminal ϕ_{j+1} or $Y_j = r_j$ for a terminal ϕ_{j+1}
- 13: Minimize Loss $(Y_j - Q(\phi_j, a_j; \theta))^2$
- 14: **end for**
- 15: **end for**

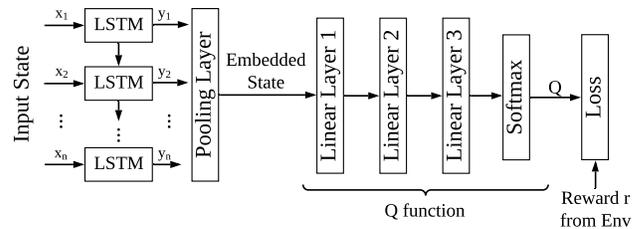


Figure 6: Neural architecture of the policy and value network.

Figure 6 shows the neural network architecture of our policy and value network. We first embed the input source code with LSTM and a pooling layer. Then, we pass the embedded state to three fully-connected layers. The Softmax function will output the Q value; the action will be selected either by maximizing the reward or with a small probability to play a random action. Finally, we calculate the loss function by the real reward r and the Q value, and update the trainable parameters θ in the Q function. Here we adopt two sets of parameters (θ and θ') in the same shape. One is for selecting an action and another one is for evaluating an action. They are updated alternatively.

3.2.4 Monte-Carlo Tree Search

Searching for correct edits is non-trivial due to two key challenges – exponentially large search space and unnecessary decision search. First, with the increase of lines of code in programs and the number of actions, the search space grows exponentially. This leads to exponential search time using uninformed search algorithms such as enumerative search [3, 4]. Second, a correct decision search requires both localize and make precise edits to generate a PM code as illustrated in

Figure 5. An invalid edit that introduces bugs leads to unnecessary decision search, if the model is unaware of such bugs and proceeds to search for new actions. To tackle the first challenge, we adopt an efficient search strategy Monte-Carlo Tree Search (MCTS) [67], which is highly efficient and effective in searching exponentially increased states. To address the second challenge, we combine the policy and value network (Section 3.2.3) with MCTS to guide selection towards potentially good states and efficiently evaluate the payout.

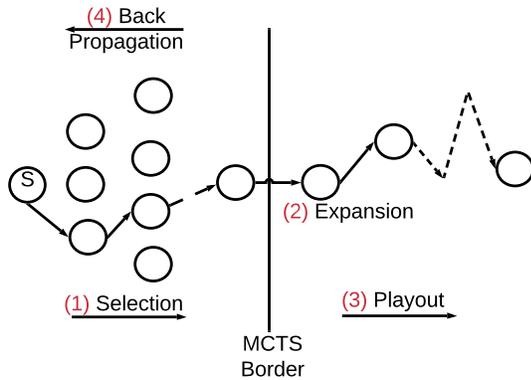


Figure 7: Overview of a simulation of Monte-Carlo tree search.

As shown in Figure 7, one round (or “simulation”) of MCTS consists of four phases. (1) **Selection** starts from the root node and selects the successive child nodes until reaching a leaf node. Here, the root node represents the current state, while a leaf node is a potential child node with at least one unexplored child (i.e., not covered by previous search). We employ a widely-used Upper Confidence Trees (UCT)-based strategy [37] to select the node with the maximum value V

$$V = q_i + c \sqrt{\frac{\ln N_i}{n_i}} \quad (3)$$

where q_i is the current action-value estimate after the i -th move; n_i is the number of simulations for the node considered after the i -th move; N_i is the total number of simulations after the i -th move run by the parent node of the one considered. c is the exploration parameter that is theoretically equal to $\sqrt{2}$; in practice, it is typically chosen empirically. The first term is high for a move that gets a high successful edits rate. The second term is high for a move with few simulations. Therefore, a large V value will lead the search to a better final solution. (2) **Expansion** unless reaches a goal state, create one or more valid child nodes. (3) **Playout** instead of evaluating the position after running a full simulation and sampling the moves until reaching a goal state in the vanilla MCTS, we approximate the value of the position by the deep Q-learning network. (4) **Back-propagation** utilizes the result from playout to update node information in reverse order.

3.2.5 Transfer Generating Knowledge of C/C++ to Java

To fully leverage the code translation knowledge learned from the RL generators for C and C++, we train a Java generator model by employing transfer learning [64]. Transfer learning is an effective technique to take features learned from one domain and leverage them in a new domain with certain similarity, to save training time and achieve higher inference accuracy on the new domain. The most commonly used transfer learning technique is fine-tuning [82], which freezes (sets as un-trainable) some layers of a pre-trained model and train the other trainable layers on a new dataset to turn the old features into predictions on the new dataset. However, the fine-tuning is a destructive process, because the new training will directly discard the previously learned function and lose the pre-trained knowledge on generating C/C++ code.

To address this issue, we employ a progressive network technique [64]. Progressive network also leverages the pre-trained models, whereas utilizing the output rather than the pre-trained parameters. Doing so overcomes the problem of discarding the prior knowledge in further training. Given a pre-trained RL generator G_j , with hidden activations $h^{(1)j} \in R^{n_i}$ from layer i , where n_i is the number of units at layer i , the progressive network utilizes a single hidden layer multi-layer perception σ to adjust different scales of various hidden layer inputs and adapt the hidden input to the new model by

$$h_i^k = \sigma(W_i^{(k)} h_{(i-1)}^k + U_i^{(k:j)} \sigma(V_i^{(k:j)} \alpha_{i-1}^{(<k)} h_{i-1}^{(<k)})) \quad (4)$$

where the $h_i^{(k)}$ is the hidden activations for layer i in task k , $W_i^{(k)}$ is the weight parameters for layer i in task k , $U_i^{(k:j)}$ are the lateral connections (a $n_i \times m_j$ matrix) from layer $i-1$ of task j to layer i of task k , $V_i^{(k:j)}$ is the projection matrix to be trained, and α is a learned scalar initialized to a small random value. The progressive network fully exploits the hidden knowledge represented by the hidden activation in multiple similar tasks in sequence.

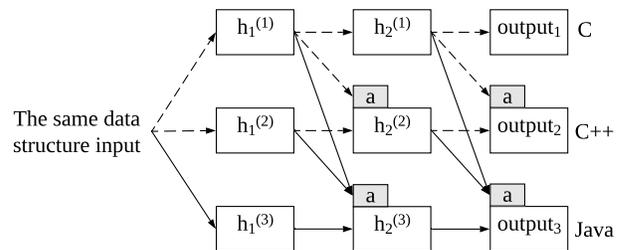


Figure 8: The workflow of the progressive network to transfer from generating C and C++ code to generating Java code, where a refers to the adaptor function shown by Equation 4.

By adopting fine-tuning and progressive networking, we take advantage of the generators for C and C++ which are well-pretrained on a sufficient training dataset and show the

potential of assisting writing PM code for other programming languages.

3.3 Code Refining Pipeline

Previous ML-based code generators pay much attention to improving the inference accuracy. Yet, even on well-investigated translation tasks (*e.g.*, CoffeeScript-JavaScript Task [14] and FOR/LAMBDA translation task [19]) on popular programming language (*e.g.*, C/C++/JAVA/Python), state-of-the-art deep learning models only achieve up to 70% ~ 90% inference accuracy [14, 19]. Due to the inherent fuzzy and approximate nature of deep learning, it is impossible to achieve 100% accuracy for all generated programs only by an ML model, let alone many complicated bugs that require programmer’s domain knowledge to detect and fix.

To address the limitations of ML-based method, we design a code refining pipeline to search for bugs that may remain in the code generated by the RL model. As shown in Figure 4, we incorporate a set of code refining tools in three levels: (1) *compilers* check on the syntax and library API usage bugs; (2) *sanity checkers*, including PMEMCHECK and PM-Reorder [17], check on the consistency and reordering bugs; (3) *validation tools*, including PMTest [45], XFDetector [44], and AGAMOTTO [54], perform a deep bug search in terms of crash consistency and failure recovery. We organize these tools as a pipeline to ensure that high-level bugs (*e.g.*, syntax bugs) are identified as early as possible before starting the time-consuming deep bug search. The output of the code refining pipeline is a code refining suggestion report for programmers to further manually inspect the code.

Figure 9 shows an example output of our code refining pipeline. In this example, the vanilla generated code is called `node_construct` followed by a `pmemobj_persist`. Our code refining pipeline identifies that `node_construct` already persists the tree node data. Therefore, there is no need to persist it again. This optimization suggestion is reported to the programmer, who decides to remove the redundant persistence API call, leading to the improved code.

```

Vanilla generated code
int node_construct() {
// set up btree node
...
pmemobj_persist(pop, node,
a->size);
// persist data
}

void btree_insert() {
// set up btree structure
...
POBJ_ALLOC(..., node_construct);
pmemobj_persist(pop, dst, args.size);
// duplicated persistence
}

Improved code
int node_construct() {
// set up btree node
...
pmemobj_persist(pop, node,
a->size);
// persist data
}

void btree_insert() {
// set up btree structure
...
POBJ_ALLOC(..., node_construct);
}

```

Figure 9: A piece of B-Tree code improved by the code refining pipeline.

4 Implementation

4.1 Training and Testing Dataset

We are the first to develop both training and testing datasets for ML-based PM code translation problems. We preprocess the datasets by including header files, creating the pool file (memory-mapped file) and initializing the root object for further read and write operations.

Training Set. We use the dataset from the PMDK library [17] with 18 C code examples, 14 C++ programs, and two Java programs. Because the PMDK example code is the nonvolatile version expert code, we obtain a volatile version of each program by manually discard the PMDK APIs in the code.

Testing Set. To show the generalization ability of our model, we test it with code never seen by the model and measure how well the model generates for such new code in the inference process. Our testing dataset consists of volatile code of 48 C programs, 42 C++ programs, and 13 Java programs obtained from the open-source Leetcode solution [6, 61, 83]. These programs perform similar functionality as PMDK example programs on various data structures widely used in persistent memory applications, including array, string, list, queue, btree, rbtree, hashmap, and combinations of multiple data structures.

4.2 Training Configuration

Deep RL Implementation. We implement the RL generator in PyTorch [58]. We identify the suitable hyper-parameters of the RL generators and update the weight parameters in the policy and value network in the training process. In particular, the LSTM encoder in our model has two recurrent layers, each with 128 cells. The *string* has on average 112 tokens, which are embedded into 48-dimensional vectors. The LSTM encoder for state embedding and the policy network are trained together in an end-to-end manner. We set the decay rate $\alpha = \frac{K}{K+T}$ to achieve the temporal decay, with $K = 10$ and T as the epoch number. We set the discount rate γ as 0.95. For the reward function, when using different checkers, the relative relationship between step, modification, and code error penalties can be different. In our model, we set step penalty factor ϕ_1 as -0.01 and modification penalty factor ϕ_2 as -0.005 . For the impact factor of each tool in the code error penalty, we set $\rho_1 = -0.1$ and $\rho_2 = -0.06$ and $\rho_3 = -0.065$ for PMEMCHECK, PM-Reorder and PMTest respectively. We set the maximum reward as 1. We train the RL generator on two Nvidia GeForce GTX 2080 Ti GPUs with 11 GB Memory for 60 epochs.

Transfer Learning Implementation. The progressive network is also implemented in PyTorch [58], and trained on two NVIDIA GeForce GTX 2080 Ti GPUs with 11GB Memory for 30 epochs. We use a dynamic learning rate scheduler in PyTorch with an initialization of 0.001 for progressive networks.

4.3 PMDK APIs

PMDK [17] is a collection of libraries developed for PM programming. These libraries build on the DAX feature that allows applications to directly load and store to PM by memory-mapping files on a PM-aware file system. We employ the *libpmobj* library, which provides a transactional object store, memory allocation, and transactions with atomicity, consistency, isolation, and durability for PM systems. We pick 21 library functions (such as `pmobj_persist`, `pmobj_create`, `pmobj_open`) and 78 macros (such as `TOID`, `POBJ_ALLOC`, `D_RW`, `D_RO`, `POBJ_ROOT`). For C++ code, we employ *libpmobj-cpp*, a C++ binding for *libpmobj*, which is more approachable than the low-level C bindings. For Java code, we use persistent collections for Java based on the *libpmobj* library.

4.4 Checkers

We use checkers in both the *environment* of our RL model and the code refining pipeline. As illustrated in Figure 4, the checkers are organized into three levels. First, we use compilers (such as `gcc`) to detect syntax errors. Second, we use sanity checkers, such as `PMEMCHECK` and `PM-Reorder`, to detect consistency and reordering bugs. Finally, we use high-level validation tools, such as `PMTest` [45], `XFDetector` [44], and `AGAMOTTO` [54], to further capture durability and ordering violations in PM operations, cross-failure bugs [44], and redundant cache line flushes and fences [54].

As demonstrated in Section 5.1, the more checkers used, the higher the PM checker passing rate and robustness. However, increasing the number and complexity of checkers also leads to a much longer training time. Therefore, we only use `PMTest` [45] in the *environment* in RL model training. The code refining pipeline adopts all three aforementioned high-level validation tools one after another to validate a program generated by our RL model. As these are independent of each other, the order of running the tools does not matter. In the following, we discuss the implementation details of integrating various checkers in *Ayudante*.

Compilers. We adopt `gcc` version 7.5.0. as the compiler in this paper.

PMEMCHECK. `PMEMCHECK` is a persistent memory analysis tool that employs the dynamic memory analysis tool `Valgrind` [55], to track all stores made to persistent memory and inform programmers of possible memory violations. Other than checking and reporting the non-persistent stores, `PMEMCHECK` also provides other options to look out for memory overwrites, redundant flushes, and provides transaction checking such as check stores that are made outside of transactions or regions that overlapped by different transactions. Here we mainly feedback the error number from the error summary reported by `PMEMCHECK` to the reward in the model.

PM-Reorder. `PM-Reorder` is another tool for persistence correctness checking. It will traverse the sequences of stores

between flush-fence barriers made by the application, and replays these memory operations many times in different combinations, to simulate the various possible ways the stores to the NVDIMM could be ordered by the system. Given an exhaustive consistency checking function, this process will uncover potential application bugs that otherwise could have been encountered only under specific system failures. It provides various optional checking orders, such as *ReorderFull* to check all possible store permutations, *ReorderPartial* to check 3 random order sequences, *ReorderAccumulative* to check a growing subset of the original sequence. Here we mainly use *ReorderPartial* to achieve consistency checking while keeping training efficiency. `PM-Reorder` requires users to provide a user-defined consistency checker, which is a function that defines conditions necessary to fulfill the consistency assumptions in source code and returns a binary value (0 represents consistent and 1 represents inconsistent). With each value in a data structure, *Ayudante* provides a default consistency checker, which determines whether each value field is properly assigned compared to the number that we store in the value field [59]. For example, if the main function sets nodes 0, 1, and 2 of a list as 100, 200, and 300, *Ayudante* will provide a consistency checker that traverses these three nodes to evaluate whether their corresponding values are correctly assigned or not. Here we mainly leverage the inconsistency number reported by `PM-Reorder`.

PMTest. `PMTest` is a fast and flexible crash consistency detecting tool, which reports violations in durability and ordering of PM operations, such as whether a persistent object has been persisted, ordering between persistent updates, and unnecessary writebacks or duplicated logs. With C/C++ source code, *Ayudante* automatically generates annotations using the C/C++-compatible software interface offered by `PMTest`, including (i) wrapping the entire code with `PMTest_START` and `PMTest_END` functions and (ii) using `TX_CHECKER_START` and `TX_CHECKER_END` to define the boundary of each transaction as required by the `PMTest` high-level checkers. These annotations will be removed after testing the generated code. We use the high-level checkers to validate three items: (1) the completion of a transaction, (2) the updates of persistent objects in the transaction are recorded in the undo log before modification, and (3) the code is free of unnecessary writebacks or redundant logs that constitute performance bugs. An issue with (1) or (2) will be reported as a `FAIL`, while issues with (3) are identified as `WARNINGS`. During the training process, we use the number of `FAILS` from `PMTest` as feedback to the neural network. In the refining pipeline, we use the `WARNING` information to suggest code refining on removing the redundant writebacks and logs.

XFDetector. `XFDetector` detects cross-failure bugs by automatically injecting failures into a pre-failure execution; it also checks cross-failure races and semantic bugs in the post-failure continuation. With C/C++ source code, *Ayudante* generates the annotations by wrapping the

Table 1: Intel server system configuration.

CPU	Intel Cascade Lake engineering sample 24 Cores per socket, 2.2 GHz 2 sockets, HyperThreading off
L1 Cache	32KB 8-way I\$, 32KB 8-way D\$, private
L2 Cache	1MB, 16-way, private
L3 Cache	33MB, 11-way, shared
TLB	L1D 4-way 64 entries, L1I 8-way 128 entries STLB 12-way 1536 entries
DRAM	DDR4, 32GB, 2666MHz, 2 sockets, 6 channels per socket
NVRAM	Intel Optane DIMM, 256 GB, 2666 MHz 2 sockets, 6 channels per socket
Kernel	Linux 5.1
Software	GCC 7.1, PMDK 1.7

code with `RoIBegin(1, PRE_FAILURE|POST_FAILURE)` and `RoIEnd(1, PRE_FAILURE|POST_FAILURE)` functions offered by XFDetector’s C/C++-compatible interface to define the region-of-interest (RoI); such annotations will be removed after code testing.

AGAMOTTO. AGAMOTTO detects persistency bugs using two universal persistency bug oracles based on the common patterns of PM misuse of C++ code: it identifies (i) modifications to PM cache lines that are not flushed or fenced; (ii) duplicated flushes of the same cache line or unnecessary fences. AGAMOTTO symbolically executes PM workloads to identify bugs without requiring annotations.

5 Evaluation

Experimental Setup. Table 1 shows configuration of the Intel server adopted in our experiments. We configure the Optane DIMMs in App Direct interleaved mode [33] and employ ext4 filesystem in DAX mode. We disable hyper threading and boost the CPUs to a fixed frequency to maintain a stable performance. We run each program 10 times and report the geometric mean.

5.1 PM Checker Passing Rate in Inference

We employ PM checker passing rate (CPR) defined in Equation 5 – the percentage of generated code that passes all the PM checkers – to measure the generalization ability of the trained model.

$$CPR = \frac{\#PassCheckers}{\#Generated} \quad (5)$$

In our experiments, we use PMEMCHECK as the checker to verify the PM CPR in inference. We train three versions of RL generators with various checker combinations in the environment, including (1) PMEMCHECK; (2) PMEMCHECK and PM-Reorder; (3) PMEMCHECK, PM-Reorder, and PMTest. separately and Table 2 shows the CPR in inference tested on microbenchmarks, a key-value store application, and Leetcode solution set. Our results show that all the

generated code of RL generator with checker combinations (2) and (3) passes the checkers with the key-value store workload testing and microbenchmarks incorporating array, string, list, queue, btree, rbtree, and hashmap data structures. Our experiments also show that Ayudante can handle programs with multiple data structures. For example, the Leetcode solution for merging k -sorted lists [7] adopts both `list` and `queue` data structures; the generated code makes both data structures persistent and passes the PMEMCHECK checker. Moreover, we observe that the more checkers used to feedback the reward, the higher CPR is achieved in inference. This is because checkers will complement each other to penalize different types of bugs and encourage more precise edits on the source code to avoid the bugs. Note that none of the existing ML-based models can achieve 100% inference accuracy due to the inherent approximate nature of ML. However, our method improves the CPR in inference by effectively taking advantage of different checkers in the training process and the refining pipeline to further report the testing and show improvement suggestions.

Table 2: The PM CPR in inference and the average percentage lines of code (LOC) changes, compared among using various checkers in the environment during training process of deep RL-based code generator.

Testing Set	Checkers in Environment	CPR	LOC
Microbenchmarks and KV store application	PMEMCHECK	87.5%	12.3%
	PMEMCHECK & PM-Reorder	100%	13.4%
	PMEMCHECK & PM-Reorder & PMTest	100%	13.8%
Leetcode solution	PMEMCHECK	60.2%	12.5%
	PMEMCHECK & PM-Reorder	62.1%	13.1%
	PMEMCHECK & PM-Reorder & PMTest	78.7%	13.4%

5.2 Execution Performance

We also evaluate the execution performance of the generated code under various input sizes by running the programs on an Intel Optane DIMM server configured as Table 1. We show the memory bandwidth performance in Figure 10a and Figure 10 on `lists`, `array`, `queue`, and `btree` microbenchmarks. As shown in Figure 10a, the generated code achieves a similar bandwidth compared with expert code provided by PMDK code examples. We also employ `perf` [25] to profile the last level cache load, store, and data TLB (dTLB) load events. As shown in Figure 10, the generated code has a cache and TLB performance comparable to PMDK. Although `lists` has a higher number of (30%) LLC store events due to unnecessary persistence API calls, it does not significantly affect the overall program execution performance.

We further evaluate the scalability of generated code by investigating the performance with different input sizes. Figure 11 shows that the bandwidth of the generated code is comparable to PMDK examples. Therefore, our code generator

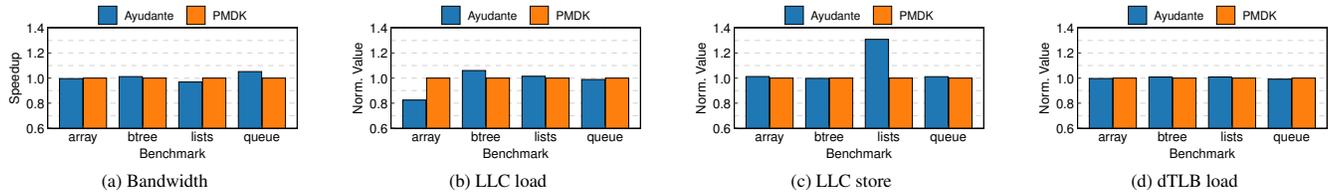


Figure 10: Performance evaluation of code generated by Ayudante, on bandwidth, LLC load, LLC store, and dTLB load on Intel server with Optane DC PM. The bandwidth is calculated based on the modified data structure size. Numbers are normalized to corresponding PMDK microbenchmark performance.

is highly stable and reliable in generating high-performance PM code.

5.3 Bug Avoidance and Reduction

Ayudante employs two mechanisms to avoid and reduce both syntax and performance bugs.

Avoidance. During the training process, we utilize the checkers to penalize our model from buggy edits to avoid bugs in the generated code. We use PMEMCHECK to penalize non-persistent stores and misused transactions. We adopt PMReorder to penalize in-consistency stores. We employ PMTest to penalize non-persistent objects, wrong ordering, and redundant logs. As an example in practice, we observe 17 non-persistent errors in an intermediate code state during the training process; after training, all the bugs are eliminated after step-by-step searches and edits. This also demonstrates that the RL generator is able to help to debug.

Reduction. Beyond the sanity checks using PMEMCHECK and PMReorder, we also design a code refining pipeline to further perform deep bug search and code refining suggestions generation. An example is shown in Section 3.3 to demonstrate the results of the code refining pipeline.

5.4 Labor Effort Reduction

We evaluate the reduction of programming labor effort with average lines of code (LOC) changed as shown in Table 2. In our experiments, the percentage of LOC changes is typically 12% ~ 15%, which significantly reduces the labor effort on developing PM applications. We test the LOC of three different versions of models by adopting different numbers of checkers. Intuitively, more checkers leads to more robust generated code, hence more APIs inserted to guarantee consistency; this results in more LOC changes. However, the difference of the total LOC is small among different versions of models, while our models bring significant CPR improvement as demonstrated in Table 2.

6 Discussion

Limitations of ML-based Approaches. Due to limitations of ML as discussed in Section 3.3, it is impossible for an ML model to achieve 100% inference accuracy with all programs due to the inherent fuzzy nature of ML [11, 14, 19]. In fact, inference accuracy improvement remains a critical challenge in ML community [23, 49]. To address the accuracy limitation,

Ayudante’s code refining pipeline effectively reduces user efforts on debugging and optimizing the generated programs.

Limitations of PM Checkers. Ayudante relies on PM checkers during model training to provide rewards and in the code refining pipeline to provide code optimization suggestions. Therefore, the capability of PM checkers is critical to the quality of code generated by Ayudante. So far, none of the existing PM checkers detects all PM bugs or programming issues. Ayudante addresses the issue by adopting a three-level checking process, which incorporates multiple PM checkers, to generate the rewards during RL model training (Section 4.4); different checkers complement each other during training. As demonstrated in Table 2, the more checkers used, the higher the PM checker passing rate and robustness. Furthermore, our code refining pipeline adopts multiple high-level validation tools, such as XFDetector [44], and AGAMOTTO [54], to further detect bugs that are not captured in training. Once more comprehensive PM checkers are developed by the community, we can retrain our model by replacing the current PM checkers to further improve PM code generation and refinement.

7 Related Work

To our knowledge, this is the first paper to design an ML-based automatic PM program assistant. This section discusses related works.

PM Programming Tools. Prior works focused on developing PM debugging and testing tools [44, 45, 54], programming libraries and APIs [10, 17, 22, 31, 74]. The tools are helpful for PM programming. However, these tools require users to manually write PM code from the scratch, which is challenging for non-expert programmers and tedious and time-consuming work for expert programmers as discussed in Section 2. Recent works also explored user-friendly libraries and high-level programming semantics for converting the data structures developed for volatile memory into PM programs [27, 42, 47]. Concurrent work TIPS [39] goes further and provides a systematic framework to convert DRAM-based indexes for the NVMM with plug-in APIs to plug-in a volatile index and facade APIs to access the plugged-in index. However, Ayudante focuses on automatically generate PM code by inserting library API functions in the source code. As such, Ayudante is orthogonal to these libraries and programming semantics;

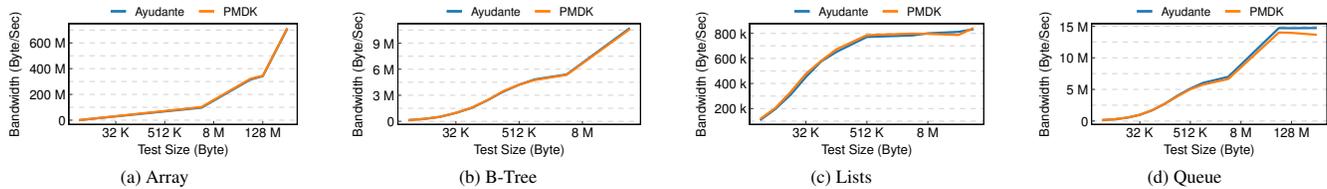


Figure 11: Bandwidth comparison between Ayudante generated code and PMDK microbenchmarks on the Intel server.

the model can also be trained based on such libraries as a substitute or complementary of PMDK.

Conventional Code Translation. Code translation problem on popular and familiar programming language, *e.g.*, C/C++/Java/C#/Python/Go, has been well investigated [2, 43, 46, 57, 71]. Previous tasks focus on adding annotation and source-to-source code translation can be categorized as 3 main folds: (1) Adding specific annotations to achieve certain constraints. An example is parallelization annotation, annotating to a program with proper statements that are required by parallel computing libraries and make the source program can run on parallel hardware, *e.g.*, OpenCL [71], CUDA [57] and OpenMP [43]. (2) translate between different types of source code. Such as source-to-source compilers LLVM [46] to translate between C and C++ by first compiling source code to LLVM bitcode then decompiling the bitcode to the target language. (3) translate code between different versions of one programming language. For example, python’s 2to3 tool [2] translates from python version 2 to version 3 by parsing the program to abstract syntax tree then translate it. Though transcompilers are preferred among the software developers for its definite transformation process, it suffers from tremendous developing labor efforts and is error-prone.

ML-based Program Generation. To model the edits to transform code into a target code, one needs to learn the conditional probability distribution of the target code version given the source code. A good probabilistic model will assign higher probabilities to plausible target versions and lower probabilities to less plausible ones. Neural Machine Translation models (NMT) are a promising approach to realize such code edit models and use distributed vector representations of words as the basic unit to compose representations for more complex language elements, such as sentences and paragraphs, *e.g.*, the sequence-to-sequence (Seq2Seq) models [35, 79] and word embedding [9, 12, 56]. However, code edits also contain structural changes, which requires the model is syntax-aware. To overcome the rigid syntax constraints in programming language, recent studies leverage tree-to-tree LSTM encoder-decoder on abstract syntax tree for program statements translation [11, 14]. However, these work are either rule-based that requires additional knowledge of the programming languages, such as grammar [11, 52, 66, 81, 84], or applying a model to implicitly learn the translation policies [14, 81] that require enough training dataset to achieve the high inference performance, which is challenging to apply to a new task without

training dataset.

8 Conclusions

We propose Ayudante, a deep reinforcement learning based framework to assist persistent memory programming. Ayudante provides a deep RL-based PM code generator that mimics programmers behavior to add proper PM library APIs to the volatile memory-based code. Ayudante also provides a code refining pipeline that reports code improvement suggestions to the programmers, to help reduce bugs and improve run-time performance. Ayudante utilize a novel transfer learning to transfer PM programming semantics from C/C++ to other languages like Java. We evaluate Ayudante with microbenchmarks of various data structures which pass all code refining pipeline checkers and achieve comparable performance to expert-handwritten code. Ayudante significantly reduce the burden of PM programming, and shed light on machine auto programming for PM and other domains.

9 Acknowledgement

We thank our shepherd Dr. Changwoo Min and the anonymous reviewers for their valuable feedback. This paper is supported in part by NSF grants 1829524, 1817077, 2011212 and SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory.

References

- [1] Ayudante source code. <https://github.com/Hanxian97/Ayudante>.
- [2] Python 2to3 migration tool, 2020. <https://github.com/erdc/python/blob/master/Doc/library/2to3.rst>.
- [3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.
- [4] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part*

- I, volume 10205 of *Lecture Notes in Computer Science*, pages 319–336, 2017.
- [5] A. Badam. How persistent memory will change software systems. *Computer*, 46(8):45–51, 2013.
- [6] begeekmyfriend. Leetcode solution in C. <https://github.com/begeekmyfriend/leetcode>.
- [7] begeekmyfriend. Leetcode solution in C++ and Python. https://github.com/begeekmyfriend/leetcode/blob/master/0023_merge_k_sorted_lists/merge_lists.c.
- [8] Bruno Bouzy and B. Helmstetter. Monte-carlo go developments. *Advances in Computer Games*, 135, 10 2003.
- [9] Nghi DQ Bui and Lingxiao Jiang. Hierarchical learning of cross-language mappings through distributed vector representations for code. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 33–36, 2018.
- [10] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages Applications*, OOPSLA ’14, page 433–452, New York, NY, USA, 2014. Association for Computing Machinery.
- [11] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. Codit: Code editing with tree-based neural machine translation. *arXiv preprint arXiv:1810.00314*, 2018.
- [12] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Long Xiong Ong. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. *IEEE Transactions on Software Engineering*, 2019.
- [13] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.
- [14] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Advances in neural information processing systems*, pages 2547–2557, 2018.
- [15] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *ASPLOS ’20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 1077–1091. ACM, 2020.
- [16] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 105–118, New York, NY, USA, 2011. Association for Computing Machinery.
- [17] Intel Corporation. Persistent memory programming. <https://pmem.io>.
- [18] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings of the 5th International Conference on Computers and Games*, CG’06, page 72–83, Berlin, Heidelberg, 2006. Springer-Verlag.
- [19] Mehdi Drissi, Olivia Watkins, Aditya Khant, Vivaswat Ojha, Pedro Sandoval, Rakia Segev, Eric Weiner, and Robert Keller. Program language translation using a grammar-driven tree-to-tree model. *arXiv preprint arXiv:1807.01784*, 2018.
- [20] 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 Ninth European Conference on Computer Systems*, EuroSys ’14, New York, NY, USA, 2014. Association for Computing Machinery.
- [21] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning. *arXiv preprint arXiv:1811.12560*, 2018.
- [22] Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. go-pmem: Native support for programming persistent memory in go. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 859–872. USENIX Association, July 2020.
- [23] Sina Ghiassian, Banafsheh Rafiee, Yat Long Lo, and Adam White. Improving performance in reinforcement learning by breaking generalization in neural networks. In *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems, AAMAS ’20, Auckland, New Zealand, May 9-13, 2020*, pages 438–446. International Foundation for Autonomous Agents and Multiagent Systems, 2020.
- [24] Hamed Gorjiara, Guoqing Harry Xu, and Brian Densky. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming*

- Languages and Operating Systems*, ASPLOS 2021, page 415–428, New York, NY, USA, 2021. Association for Computing Machinery.
- [25] Brendan Gregg. Linux perf examples, 2019. <http://www.brendangregg.com/perf>.
- [26] Rahul Gupta, Aditya Kanade, and Shirish Shevade. Deep reinforcement learning for syntactic error repair in student programs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 930–937, 2019.
- [27] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Mod: Minimally ordered durable datastructures for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20*, page 775–788, New York, NY, USA, 2020. Association for Computing Machinery.
- [28] Alan Harris. Distributed caching via memcached. In *Pro ASP. NET 4 CMS*, pages 165–196. Springer, 2010.
- [29] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.
- [30] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [31] Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.
- [32] Terry Ching-Hsiang Hsu, Helge Brügger, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys '17*, page 468–482, New York, NY, USA, 2017. Association for Computing Machinery.
- [33] Intel. Intel® Optane™ DC Persistent Memory, 2019.
- [34] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman 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. *CoRR*, abs/1903.05714, 2019.
- [35] Yonghae Kim and Hyesoon Kim. A case study: Exploiting neural machine translation to translate cuda to opencl. *arXiv preprint arXiv:1905.07653*, 2019.
- [36] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.
- [37] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.
- [38] Petar Kormushev, Sylvain Calinon, and Darwin G Caldwell. Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, 2(3):122–148, 2013.
- [39] R. Madhava Krishnan, Wook-Hee Kim, Fu Xinwei, Sumit Kumar Monga, Hee Won Lee, Jang Minsung, Ajit Mathew, and Changwoo Min. Tips: Making volatile index structures persistent with DRAM-NVMM tiering. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.
- [40] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High performance metadata integrity protection in the WAFL copy-on-write file system. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 197–212, Santa Clara, CA, February 2017. USENIX Association.
- [41] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chausot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.
- [42] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19*, page 462–477, New York, NY, USA, 2019. Association for Computing Machinery.
- [43] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4):101–110, 2009.
- [44] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wensch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20*, page 1187–1202, New York, NY, USA, 2020. Association for Computing Machinery.
- [45] 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 Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 411–425, New York, NY, USA, 2019. Association for Computing Machinery.
- [46] LLVM. The llvm compiler infrastructure, 2020. <https://llvm.org/>.
- [47] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and fast persistence for volatile data structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 789–806, New York, NY, USA, 2020. Association for Computing Machinery.
- [48] Amirsaman Memaripour and Steven Swanson. Breeze: User-level access to non-volatile main memories for legacy software. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 413–422. IEEE, 2018.
- [49] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.
- [50] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
- [51] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.
- [52] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *International Conference on Learning Representations*, 2018.
- [53] Karthik Narasimhan, Tejas Kulkarni, and Regina Barzilay. Language understanding for text-based games using deep reinforcement learning. *arXiv preprint arXiv:1506.08941*, 2015.
- [54] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1047–1064. USENIX Association, November 2020.
- [55] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100. ACM, 2007.
- [56] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. Mapping API elements for code migration with vector representations. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 756–758. IEEE, 2016.
- [57] Cedric Nugteren and Henk Corporaal. Introducing 'bones' a parallelizing source-to-source compiler based on algorithmic skeletons. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 1–10, 2012.
- [58] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.
- [59] pmdk. pm-reorder consistency checker. <https://pmem.io/2019/02/04/pmreorder-basics.html>.
- [60] Athanasios S Polydoros and Lazaros Nalpantidis. Survey of model-based reinforcement learning: Applications on robotics. *Journal of Intelligent & Robotic Systems*, 86(2):153–173, 2017.
- [61] pozy. Leetcode solution in C++ and Python. <https://github.com/pezy/LeetCode>.
- [62] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Iron file systems. *SIGOPS Oper. Syst. Rev.*, 39(5):206–220, October 2005.
- [63] Andy Rudoff. Persistent memory programming. *Login: The Usenix Magazine*, 42(2):34–40, 2017.
- [64] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.
- [65] Akanksha Rai Sharma and Pranav Kaushik. Literature survey of statistical, deep and reinforcement learning in natural language processing. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 350–354. IEEE, 2017.

- [66] Hui Shi, Yang Zhang, Xinyun Chen, Yuandong Tian, and Jishen Zhao. Deep symbolic superoptimization without human knowledge. In *International Conference on Learning Representations*, 2019.
- [67] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.
- [68] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dhharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.
- [69] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- [70] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning*, 4(1):1–103, 2010.
- [71] Krishnahari Thouti and SR Sathe. A methodology for translating c-programs to opencl. *International Journal of Computer Applications*, 82(3), 2013.
- [72] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.
- [73] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 91–104, New York, NY, USA, 2011. Association for Computing Machinery.
- [74] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News*, 39(1):91–104, 2011.
- [75] William Yang Wang, Jiwei Li, and Xiaodong He. Deep reinforcement learning for nlp. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts*, pages 19–21, 2018.
- [76] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 496–508, 2020.
- [77] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. An early evaluation of Intel’s Optane DC persistent memory module and its impact on high-performance scientific applications. In *SC*, 2019.
- [78] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.
- [79] Shaofeng Xu and Yun Xiong. Automatic generation of pseudocode with attention seq2seq model. In *2018 25th Asia-Pacific Software Engineering Conference APSEC*, pages 711–712. IEEE, 2018.
- [80] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.
- [81] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017.
- [82] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3320–3328, 2014.
- [83] yuanguangxin. Leetcode solution in Java. <https://github.com/yuanguangxin/LeetCode>.
- [84] Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William Byrd, Matthew Might, Raquel Urtasun, and Richard Zemel. Neural guided constraint logic programming for program synthesis. In *Advances in Neural Information Processing Systems*, pages 1737–1746, 2018.
- [85] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 421–432, 2013.

TIPS: Making Volatile Index Structures Persistent with DRAM-NVMM Tiering

R. Madhava Krishnan Wook-Hee Kim Xinwei Fu Sumit Kumar Monga
Hee Won Lee* Minsung Jang† Ajit Mathew‡ Changwoo Min
Virginia Tech Samsung Electronics* Peraton Labs†

Abstract

We propose TIPS— a framework to systematically make volatile indexes persistent. TIPS neither places restrictions on the concurrency model nor requires in-depth knowledge of the volatile index. TIPS relies on novel *DRAM-NVMM tiering* to support an index-agnostic conversion, durable linearizability and its concurrency model called the *tiered concurrency* to achieve a good performance, scalability. TIPS proposes a hybrid low overhead logging technique called the *UNO logging* to guarantee crash consistency and to handle persistent memory leaks across crashes. We converted seven volatile indexes with different concurrency models and the Redis key-value store application using TIPS and evaluated them using YCSB. Our evaluations show that TIPS-enabled indexes outperform the state-of-the-art index conversion techniques PRONTO, NVTraverse, RECIPE, and the NVMM-optimized B+Tree indexes (BzTree, FastFair), Hash indexes (CCEH and Level Hash) and Trie (WOART) indexes by 3-10× while supporting strong consistency and index-agnostic conversion.

1 Introduction

Indexes are a fundamental building block in many storage systems, and it is critical to achieving high performance and reliability [34, 47]. With the advent of Non-Volatile Main Memory (NVMM), such as Intel Optane DC Persistent Memory [10, 52], there have been a numerous number of research efforts targeted towards developing NVMM-optimized indexes [5, 11, 13, 26, 30, 36, 42–44, 53–56, 58, 59, 63, 65, 66]. Such index designs mainly focus on reducing the crash consistency overhead by primarily relying on index-specific optimizations to improve the overall performance.

However, maturing and hardening an index requires a lot of time and effort. For example, recently proposed NVMM indexes have critical limitations, such as (1) weaker consistency guarantee [23, 37], (2) not handling memory leaks in the wake of a crash [11, 26, 36, 53, 66], (3) limited concurrent access [13, 27, 36], and (4) not supporting variable-length keys [13, 26, 55, 65]. Most of these missing features are critical in real-world systems and it delimits the adoption of these indexes to the real-world applications without further maturing.

Alternatively, there are decades of research on in-memory DRAM indexes [16, 39, 45, 47, 60] which are well optimized, engineered and used in many real-world applications such as

in-memory key-value stores and databases [18, 19, 32, 57]. If we can leverage these in-memory DRAM indexes for NVMM, it not only gives a large pool of well-engineered indexes but it will also pave way for the real-world applications built on top of these indexes to use and adopt NVMM. The challenges in building an NVMM-optimized index has lead to a spike in the interest to port legacy DRAM applications, particularly in-memory key-value stores [1–4, 6, 7, 46, 49]. However, prior works [6, 46, 49] report that manual porting is complex, and error-prone requiring a lot of engineering effort. So we believe that *it is important to provide a systematic way to convert DRAM-based indexes for the NVMM.*

A few recent studies have proposed techniques [50, 62] or guidelines [15, 21, 23, 37] to convert volatile indexes to NVMM. Unfortunately, these techniques have some critical limitations such as (1) limited applicability due to restrictions on the concurrency control (*e.g.*, supporting only lock-free indexes) [15, 21, 23, 37, 50], (2) supporting a weaker consistency guarantee (*i.e.*, buffered durable linearizability) [23, 37, 62], (3) requiring in-depth index-specific knowledge [15, 21, 23, 37], (4) high performance and storage overhead due to their crash consistency mechanism [23, 50, 62], and (5) not addressing persistent memory leaks [15, 21, 23, 37, 62]. We further discuss the limitations of these techniques in §2.

To address these problems, we propose TIPS— an index-agnostic framework to systematically make a volatile DRAM index persistent, while supporting (1) wider applicability by not placing any restrictions on the concurrency model of an index, (2) strong consistency model (*i.e.*, durable linearizability), (3) index-agnostic conversion without requiring in-depth knowledge on a DRAM index, (4) low overhead crash consistency mechanism, (5) safe persistent memory management (*e.g.*, no persistent memory leak), and in addition to achieving (6) high performance and good multicore scalability. This paper makes the following contributions:

- We propose a novel *DRAM-NVMM data tiering approach* and its concurrency model called the *tiered concurrency* to achieve good performance, scalability, and applicability.
- We propose a low overhead hybrid logging technique called the *UNO logging* to guarantee crash consistency, to prevent memory leaks and to guarantee durable linearizability.
- We propose the TIPS *framework* based on these approaches. TIPS provides index-agnostic conversion and does not place any restrictions on the concurrency model or require in-depth knowledge of the volatile index being converted.

*†The authors contributed to this work while they were at AT&T Labs Research. ‡The author is currently in Amazon.

- We converted seven volatile indexes with different concurrency models, a real-world key-value store Redis and evaluated them using YCSB [14]. Our evaluation shows that TIPS outperforms the state-of-the-art index conversion techniques and the NVMM-optimized indexes by 3-10× across the different YCSB workloads.

2 Background and Motivation

We discuss the limitations of existing conversion techniques and their implications below:

(1) Restrictions on Concurrency Control. All prior techniques have limited applicability; *i.e.*, they are designed to support only a specific concurrency model. For example, NVTraverse [21] and link-and-persist [15] are designed for lock-free indexes while MOD [23] is designed for purely functional data structures. PRONTO [50] is applicable only to the globally blocking indexes (*e.g.*, protected by a global mutex) while RECIPE [37] guidelines apply only to the indexes that support lock-free or fine-grained locking.

(2) Supporting Weak Consistency. Another limitation is that most techniques [23, 37, 62] support only a weaker consistency; A linearizable DRAM index converted using these techniques will support only a weaker consistency model, Buffered Durable Linearizability (BDL) [29]. Linearizability has been the standard consistency model in DRAM indexes for almost three decades [25]. Its NVMM counterpart is Durable Linearizability (DL) [29], and it plays a critical role in ensuring the correctness and consistency of the NVMM index and data. Indexes with a BDL guarantee can experience a large amount of data loss in the wake of a crash. Moreover, it increases the programming complexity as the developers are burdened with reasoning about the data consistency. This makes the conversion process complex and more error-prone; for instance, many fundamental and non-trivial crash consistency bugs have been found in the RECIPE indexes [21, 22].

(3) Requiring In-depth Knowledge. Many techniques [15, 21, 23, 37] require in-depth knowledge of the volatile index and expertise in NVMM programming to apply their guidelines correctly. Such efforts are non-trivial as even missing a single sfence or a clwb may render an index irrecoverable.

(4) High Storage and Performance Overhead. Many techniques suffer from high storage and performance overhead [23, 50, 62] mainly due to their crash consistency mechanism. For example, PMThreads [62] requires two full replicas (one each on DRAM and NVMM) of the original data. MOD [23] uses Copy-on-Write (CoW) to guarantee crash consistency. Such a high storage overhead might be acceptable for primitive data structures (*e.g.*, stack, vector). However, it can be detrimental for indexes and real-world key-value stores designed to handle a large volume of data. Although PRONTO [50] uses operational logging for crash consistency and employs a dedicated background thread for every writer to perform the logging, it still incurs a high overhead due to

the synchronous waiting between the writers and background threads. We further empirically analyze these overheads in §7.

(5) Persistent Memory Leaks. Another critical but a largely understated problem is the persistent memory leaks. While the prior conversion techniques and the NVMM-optimized indexes focus on providing crash consistency, they completely ignore the memory leak problem. Although NVMM allocators can guarantee failure atomicity for allocation/free even the mature allocator such as the PMDK [28] does not provide an efficient solution to identify and fix the memory leak [17]. Hence it is critical to address this within the TIPS framework.

3 Overview of TIPS

We first discuss our design goals followed by the design overview. We assume that an index being converted has key-value store style operations such as insert, delete, update, lookup, and scan. Throughout this paper, we address insert, delete and update as writes, and lookup and scan as reads. The term *plugged-in index* denotes a user-defined volatile index on NVMM that is plugged into TIPS framework. We assume that locks can be reinitialized after crash recovery.

3.1 Design Goals

G₁ Support Various Concurrency Models. We aim not to place restrictions on the concurrency model of the volatile indexes. With TIPS, we support the conversion of indexes that use a global lock (*e.g.*, Mutex), lock-free (*e.g.*, CAS), and fine-grained locking (*e.g.*, ROWEX [39]).

G₂ Support Durable Linearizability (DL). The challenge to guarantee DL in TIPS is to achieve it without compromising the performance, scalability, and index-agnostic conversion.

G₃ Support Index-agnostic Conversion. We aim to support near black-box, index-agnostic conversion to circumvent the need for in-depth knowledge on the volatile index and NVMM programming. This will make the conversion process simple, intuitive, and less error-prone. We aim to achieve this by internally handling the complications of NVMM programming such as guaranteeing crash consistency and preventing persistent memory leak within TIPS and also providing an uniform programming interface to assist the conversion.

G₄ Design a Low Overhead Crash Consistency mechanism. TIPS can not rely on index-specific optimizations for crash consistency to support an index-agnostic conversion. The crash consistency mechanism should incur low overhead to achieve high performance and scalability. A crash consistency mechanism also should prevent persistent memory leaks by reclaiming the unreachable objects upon recovery.

G₅ Achieve High Performance and Scalability. Ideally, we aim to perform and scale on par with NVMM-optimized indexes. Also, we strive to retain the original characteristics of the plugged-in index; for example, if the volatile index is designed for cacheline efficiency or optimized for scalability, we aim to retain and leverage such characteristics to improve

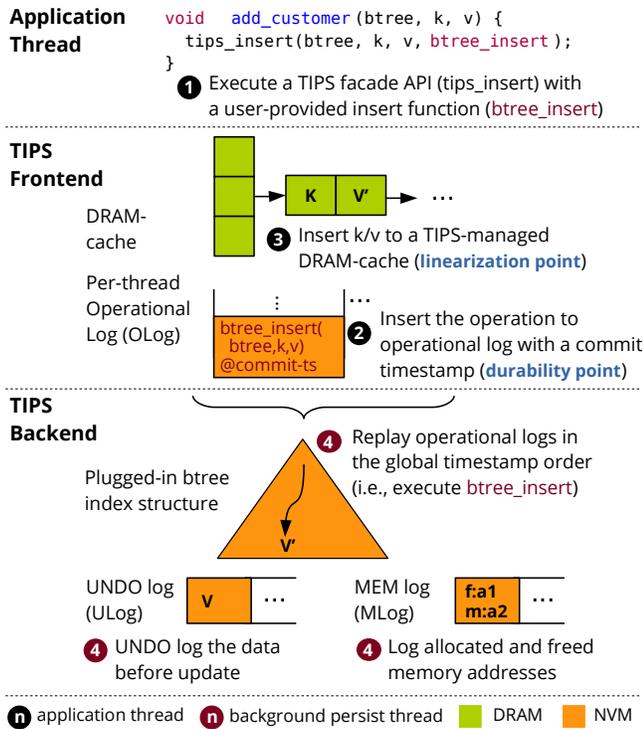


Figure 1: Illustrative example of inserting a key-value pair in TIPS.

the overall performance and scalability of a TIPS index.

3.2 Design Overview

3.2.1 High-Level Idea of TIPS

Figure 1 presents the TIPS architecture and illustrates how a write operation is handled in the TIPS framework. TIPS frontend consists of a hash table on DRAM (DRAM-cache) and an operational log (OLog) on NVMM. TIPS backend consists of the plugged-in index – the user-provided volatile index – a background thread (persist-thread), UNDO log (ULog), and MEM log (MLog). When a write is issued (①), TIPS first commits it to the per-thread OLog for guaranteeing durability (② - durability point) and then inserts a new key-value pair entry (or a tombstone for deletion) in the DRAM-cache to make the write visible (③ - linearization point). Then the persist-thread in the TIPS backend (④) replays the same write in the background to update the plugged-in index. To guarantee crash consistent update to the plugged-in index, TIPS uses ULog to store the unmodified data for recovery. Once the plugged-in index is updated, the corresponding key-value entry in the DRAM-cache will be reclaimed later.

Alternatively, a lookup operation first looks for the target key in DRAM-cache and it goes to the plugged-in index only if the target key is not present in DRAM-cache. With this high-level idea, we next present the design overview of TIPS and also explain how it contributes towards achieving our design goals discussed in §3.1.

3.2.2 DRAM-NVMM Tiering (G_1, G_2, G_3, G_5)

At its core, TIPS adopts a novel DRAM-NVMM data tiering approach. As illustrated in Figure 1, two critical components that enable the DRAM-NVMM tiering are DRAM-cache in the TIPS frontend and plugged-in index in the TIPS backend. Although prior NVMM-optimized B+tree index designs [42, 55] and conversion techniques [50, 62] have proposed to use both NVMM and DRAM, our approach is fundamentally different. For example, PRONTO [50] builds the index on DRAM and logs the index operations on the NVMM to guarantee durability. Such a tiering design limits PRONTO indexes from scaling beyond the DRAM capacity. Instead, in TIPS, we propose tiering of the data (i.e., key-value pairs) while keeping the plugged-in index intact on the NVMM.

Benefits. (1) Tiering the data between DRAM-NVMM makes our approach *generally applicable* to any index. This enables all the TIPS indexes to take advantage of the faster DRAM. (2) Unlike PRONTO, TIPS can achieve a better capacity scaling by keeping the plugged-in index on the NVMM. (3) The writes are made visible through DRAM-cache (③ in Figure 1); this enables TIPS to guarantee DL agnostic of the plugged-in index. (4) On top of DRAM-NVMM tiering, we build the *plug-in programming model* to enable index-agnostic conversion (§3.2.6). (5) Tiering the data enables a new concurrency model called the *tiered concurrency*, which is key to achieving high performance and scalability.

3.2.3 Tiered Concurrency for Scaling Frontend (G_1, G_5)

Having TIPS frontend and backend enables two different levels of concurrency: (1) concurrency model of the DRAM-cache and (2) concurrency model of the plugged-in index. We call this a *tiered concurrency model*. TIPS frontend allows parallel readers and parallel disjoint writers as DRAM-cache is a concurrent hash table and OLog is per-thread. In the critical path, all requests succeeding in the TIPS frontend (all writes and read hits) will follow the concurrency model of DRAM-cache, and the operations that go to the TIPS backend (read misses and scan) will follow the concurrency model of a plugged-in index. Also, the background writes (④ in Figure 1) to the plugged-in index is done off the critical path adhering to concurrency model of the plugged-in index.

In a nutshell, to achieve write scalability TIPS restricts writes to the faster frontend (DRAM-cache and per-thread OLog) and for read scalability, it relies on both the DRAM-cache and the concurrency model of the plugged-in index. Range scans always go to the plugged-in index as it requires full key-value data (see details in §4.1.6). Relying on the plugged-in index for read/scan helps TIPS to reduce the DRAM footprint as it does not need to cache the entire dataset in DRAM-cache. Instead, it can reclaim the keys once the plugged-in index is updated.

Benefits. (1) Even if the plugged-in index supports only blocking concurrency (e.g., mutex), it can still leverage the

```

1 /* TIPS facade API to use a TIPS-enabled index */
2 bool tips_insert(void *ds, key_t k, value_t v, fn_insert_t *f);
3 bool tips_update(void *ds, key_t k, value_t v, fn_update_t *f);
4 bool tips_delete(void *ds, key_t k, fn_delete_t *f);
5 value_t *tips_lookup(void *ds, key_t k, fn_lookup_t *f);
6 value_t *tips_scan(void *ds, key_t start_key, int range,
7                   fn_scan_t *f);
8
9 /* TIPS plug-in API to implement a TIPS-enabled index */
10 bool tips_u_log_add(void *addr, size_t size);
11 void *tips_alloc(size_t size);
12 void tips_free(void *addr);

```

Figure 2: TIPS facade APIs to access a TIPS-enabled index, and plug-in APIs for plugging-in a volatile index to TIPS.

DRAM-cache to process all the writes and read hits concurrently to achieve good performance. (2) Unlike the previous techniques [15, 21, 37, 50], it does not place any restrictions on the concurrency model of the volatile indexes and hence any volatile index can be plugged-in to the TIPS framework.

3.2.4 Adaptive Scaling for Backend Scalability (G_5)

The backend writes are slower than the frontend as it writes to the NVMM. This can cause an imbalance in the system; to prevent this, we propose adaptive scaling of background writers (workers) for scaling the TIPS backend. With adaptive scaling, TIPS continuously monitors both the frontend and backend write throughput, and when there is an imbalance it scales up the worker count to catch up with the faster frontend and vice versa. While scaling up, TIPS identifies the best worker count based on the write scalability of the plugged-in index and it caps the scale-up at that count to get maximum performance. Also, the real-world workloads are rarely 100% writes, so the time between writes and the adaptive scaling can help the backend writers to catch up with a faster frontend.

Benefits. (1) It effectively utilizes the write concurrency of the plugged-in index. (2) Unlike PRONTO [50] which demands a dedicated worker for every foreground writer, TIPS can dynamically adjust the worker count based on nature of the workload and the plugged-in index.

3.2.5 UNO Logging for Crash Consistency (G_4)

To achieve a low overhead index-agnostic crash consistency, we propose the *UNO logging protocol*, which makes a hybrid and synergistic use of traditional UNDO logging and Operational logging. In TIPS, we use operational logging (OLog) to guarantee immediate durability and UNDO logging (ULog) to ensure failure-atomicity while updating the plugged-in index. The unique aspect of *UNO* logging lies in how we leverage the OLog to reduce the notorious UNDO logging overhead and also reduce the number of p-barrier (c1wbs followed by sfence) by batching the recurring updates to the same cache line and consequently achieve a low overhead crash consistency. Furthermore, MEM Log (MLog) internally logs all the allocated and freed addresses and TIPS uses this information to identify and free all the unreachable memory upon recovery to prevent memory leaks and defers the actual memory free operations until OLog entries are

```

1 void hash_insert(hash_t *hash, key_t key, val_t value) {
2     node_t **pprev_next, *node, *new_node;
3     int bucket_idx;
4     pthread_rwlock_wrlock(&hash->lock);
5     // Find a node in a collision list
6     // Case 1: update an existing key
7     if (node->key == key) {
8         // Before modifying the value, backup the old value
9 +     tips_u_log_add(&node->value, sizeof(node->value));
10        node->value = value; // then update the value
11        goto unlock_out;
12    }
13    // Case 2: add a new key
14    // Allocate a new node using tips_alloc
15 +    new_node = tips_alloc(sizeof(*new_node));
16    new_node->key = key; new_node->value = value;
17    new_node->next = node;
18    // Backup the prev node before modifying it
19 +    tips_u_log_add(pprev_next, sizeof(*pprev_next));
20    *pprev_next = new_node; // then update then the node
21    unlock_out:
22    pthread_rwlock_unlock(&hash->lock);
23 }

```

Figure 3: Code snippet of a TIPS-enabled hash table insert. Only three lines are modified in the original code; Lines 9, 19 for UNDO logging and Line 15 for persistent memory allocation.

consumed to prevent double-free bugs.

Benefits. (1) Using OLog requires only *two p-barriers* in the critical path for all write operations; *one p-barrier* to persist a OLog record and *one* more to persist the tail pointer (§4.1.5). This makes the durability guarantee cheap and consequently a better performance (§7.2, §7.4). (2) TIPS alleviates the UNDO logging overhead by leveraging the OLog information to selectively log only the memory required for correct recovery besides the merit that UNDO logging in TIPS is performed by the background workers (§4.3.2). (3) With MLog TIPS handles the persistent memory leaks with its framework instead of delegating it to the users.

3.2.6 Plug-In Programming Model for Index-agnostic Conversion (G_2)

The plug-in programming model provides two sets of APIs as shown in Figure 2: (1) plug-in APIs to plug-in a volatile index to the TIPS framework and (2) facade APIs to access the plugged-in index. The facade APIs internally manage the OLog and DRAM-cache without requiring any user intervention (steps ①, ②, ③ in Figure 1). With the facade APIs, the plugged-in index implementation should be passed as a function pointer *f* which is used by TIPS to update the plugged-in index. To guarantee crash consistent updates to the plugged-in index, the developers must modify their index implementation using TIPS plug-in APIs. The modifications are simple, as shown in Figure 3: (1) replacing the volatile memory allocation (and free) with `tips_alloc` (and `tips_free`) and (2) adding `tips_u_log_add` before modifying/writing to an NVMM address. TIPS will execute this modified code (e.g., `hash_insert` in Figure 3) during the background reply.

By replacing the `malloc` with `tips_alloc`, the plugged-in index will now be allocated on the NVMM, and `tips_alloc` will also capture all the newly allocated address in the MLog to prevent persistent memory leaks. The developer

added `tips_u_log_add` would ensure crash consistency while updating the plugged-in index, and TIPS internally optimizes the UNDO logging (§4.3.2) for better write coalescing and performance. Moreover, a developer is not required insert p-barriers to the volatile codebase manually. Instead, they just need to annotate the stores to NVMM with `tips_u_log_add`. Thus, LoC changes in the plugged-in indexes are minimal, as shown in Table 2. While the developers still have to add the `tips_u_log_add` manually, they need not handle the persistence/visibility order as in the case of manually inserting p-barriers, and this makes the conversion easy and less error-prone. Note that updates to the newly allocated addresses (in Memlog) and existing addresses (in ULog) are batched and persisted internally by TIPS before reclaiming the respective logs. More details on this in §4.3.

4 Design of TIPS

We first describe the TIPS frontend design in §4.1, followed by the TIPS backend design in §4.2.

4.1 TIPS Frontend Design

4.1.1 DRAM-cache

The DRAM-cache is a concurrent open chaining hash table. Concurrent writers working on the different buckets are allowed to proceed in parallel while the ones working on the same bucket are synchronized using a spinlock. Readers do not need any synchronization (*i.e.*, lock-free reads) as all writes to DRAM-cache are performed via a single atomic store. DRAM-cache also employs a RCU-style epoch based reclamation scheme for garbage collection. We choose open chaining hash table to guarantee $O(1)$ writes and avoid expensive rehashing in the critical path. We discuss the configurations for chain length and bucket size in §6 and §7.

4.1.2 Handling Write Operations

All write operations are first committed to the OLog to guarantee durability, and then a new entry is created and added to the DRAM-cache to make the writes visible. For delete, the entry also carries a tombstone mark for the readers to identify that it has been deleted logically. To make writes faster, the entries are always added at the head of a bucket. We explain how TIPS guarantees Durable Linearizability (DL) in §5.

4.1.3 Handling Lookup Operation

Readers first traverse the DRAM-cache bucket looking for the target key. As the collision chain is sorted by the arrival order of write requests, the readers can stop at the first match instead of traversing the entire chain. If the key is not present, then TIPS internally redirects the readers to the plugged-in index using the function pointer provided in the `tips_lookup` call. Scan operations require traversing the plugged-in index and the OLog to return a consistent result. We further describe it in §4.1.6 after introducing the OLog design.

4.1.4 Safe Reclamation

Since all writes happen in the DRAM-cache, the collision chain can proliferate and result in a high chain traversal overhead. To address this, we employ a background garbage collector thread called the gc-thread. When the chain length of a bucket exceeds the preset threshold, the gc-thread is triggered, which then visits the respective bucket and safely reclaims the entries. To ensure safe reclamation of entries *i.e.*, without impacting the concurrent readers, TIPS employs an epoch-based reclamation scheme, which is widely used in lock-free and RCU-based data structures [20, 24, 48, 61]. TIPS manages two types of epochs: (1) local epoch, which is the global epoch value when readers enter the critical section, and (2) global epoch, which is advanced when all active readers in the current epoch exit the critical section.

The gc-thread first logically reclaims entries by unlinking them from the collision chain and storing them in the free-list. The reclaimed entry then becomes invisible to new readers. Note that the gc-thread logically reclaims the entries that are successfully replayed, so upon a read miss, readers can still retrieve the reclaimed entries from the plugged-in index. To determine if there are any outstanding readers on the logically reclaimed entries, the gc-thread checks the local epoch of all the active readers. If the local epoch is the same as the global epoch, *i.e.*, no outstanding readers from the previous epoch exist; then the gc-thread physically frees the entries in the free-list that are logically reclaimed two epochs ago. Note that the gc-thread atomically modifies the collision chain, so the readers and writers are free to enter the critical section without waiting for reclamation to finish.

4.1.5 Operational Log (OLog)

OLog guarantees durability to the write operations executed on the DRAM-cache. An OLog record consists of the operation type (insert, delete or update), the respective function pointer to the plugged-in index logic, key, value, and a global timestamp (`commit-ts`) to denote the commit order of an operation. OLog is a circular buffer where new entries are always added at the tail. The atomicity for an OLog write is guaranteed by its tail pointer update. Once an OLog record is written and persisted, the tail pointer is atomically updated to point to the new record, followed by persisting the tail pointer.

4.1.6 Handling the Scan Operations

Algorithm. The scan operation in TIPS is always directed to the plugged-in index, as it requires a full range of keys and values. However, the plugged-in index is not guaranteed to be up-to-date since the persist-thread might still be propagating some of the updates that might potentially fall within the scan range. So, after scanning the plugged-in index, TIPS traverses the OLogs looking for any potential keys that might fall within the scan range. Upon finding any, the scan buffer is adjusted to incorporate not yet propagated operations.

Traversing OLog. To minimize the OLog traversal over-

head, we leverage the commit timestamp of each OLog record (`commit-ts`) and the timestamp when a scan operation starts (`scan-ts`). While traversing the OLog, the scan thread compares its `scan-ts` with the `commit-ts`. If the `commit-ts` \leq `scan-ts`, then the scan thread reads the key information in the OLog record and checks if it falls within the scan range. It is safe to ignore OLog records with `commit-ts` $>$ `scan-ts` because these are in the future with respect to the scan thread, so it can stop traversing the OLog. Refer to §5 for correctness. Traversing the OLog is fast and adds only a negligible overhead for three reasons: (1) We partially traverse not-yet-propagated OLog entries stopping at the first future entry. (2) As the persist-thread continuously propagates the updates, there will not be much backlog in the OLog. (3) Finally, the scanning of OLog is a sequential read operation on the NVMM, which is almost as fast as reading from the DRAM [64]. We verify this empirically in §7.3.

4.2 TIPS Backend Design

The primary role of the TIPS backend is to combine and replay the per-thread OLog on the plugged-in index. To guarantee correct replay order, the persist-thread combines the per-thread OLog entries, and then it spawns the required number of background workers, which replays the combined entries to the plugged-in index. The persist-thread decides the required worker count on-fly using the adaptive scaling algorithm. The following subsections explain each of these steps in detail.

4.2.1 Adaptive Scaling of Background Workers

TIPS automatically adjusts the number of workers at every epoch based on the write scalability of the plugged-in index and the nature of the workload. One epoch (e) is defined as one iteration of combining and replying the OLog entries. We denote W_{e+1} as the worker count for the next epoch $e + 1$.

At every epoch e , TIPS calculates the foreground throughput (F_e), which is the number of OLog entries produced by application threads during the epoch, and the background throughput, which is the number of OLog entries consumed by the worker threads during the epoch. TIPS calculates the processing rate R_e , which is F_e/B_e . It aims to maintain R_e close to 1 by adjusting the number of workers (W_{e+1}).

If $R_e > 1$, the foreground writers are filling up the OLog at a faster pace, and if this situation persists, it will lead to blocking of writers as the workers are slow in clearing up the OLogs. So TIPS will increase W_{e+1} by a predefined step Δ aiming to improve the B_e and keep up with F_e .

If $R_e < 0.5$, workers are clearing up the OLogs faster than foreground writers fill them up. Employing excess number of workers is a waste of CPU, so TIPS decreases W_{e+1} by Δ .

Finally, if $1 < R_e < 0.5$, TIPS considers that the workers are on par with foreground writers and hence maintains the same number of workers (*i.e.*, $W_{e+1} = W_e$).

TIPS maintains a user-configurable upper bound (W^{up}) and a lower bound (W^{low}) to cap the number of workers (W_e)

while scaling up and down respectively. In addition, while scaling up, TIPS memorizes the best performing worker count (W^{max}); so that if W_e reaches the upper bound TIPS can fall back to W^{max} and continue until scaling down is needed.

By default, TIPS sets W^{low} to 0 and W^{up} to the number of physical cores. TIPS uses a smaller Δ when the worker count is small (*i.e.*, $\Delta = 1$ when $W_e < 4$) and uses a bigger Δ when the worker count is large (*i.e.*, $\Delta = 4$ when $W_e \geq 4$).

4.2.2 Concurrent Replay of OLog Entries

After deciding the number of workers, persist-thread combines the per-thread OLog entries and adds them to the per-worker queue. To avoid copy overhead, TIPS maintains only a pointer to the OLog record in per-worker queue.

There are two key invariants that must be maintained in the combining process: (1) Since the OLog records are replayed concurrently, it is essential to maintain the correct ordering, especially for non-commutative operations (*e.g.*, `insert(k1, v)` and `delete(k1)`). For commutative operations (*e.g.*, `insert(k2, v)` and `delete(k3)`), the replay can be done in any order without violating the correctness. (2) The OLog can not be reclaimed until all the entries in the per-worker queues are consumed. For (1), TIPS uses hashing to ensure that all the non-commutative operations will be placed in the same worker-queue. After combining, persist-thread spawns W_e workers. Then each worker sorts the entries in the timestamp (`commit-ts`) order and replays them to the plugged-in index. This ensures that non-commutative operations are always executed by the same worker in their exact order of arrival. For (2), persist-thread waits until the end of the current epoch to safely reclaim the OLog (see the details in §4.3.3).

4.3 UNO Logging

In this section, we describe the design of ULog and MLog. Similar to OLog, the atomicity of ULog and MLog writes is guaranteed by atomic tail pointer update. Both ULog and MLog are protected using a global Readers-Writer lock.

4.3.1 Memory Log (MLog)

What to log? All the newly allocated and freed addresses are recorded in the MLog along with a tag to denote if the address is allocated or freed. Also, each allocated address carries a timestamp (`alloc-ts`) to denote the time at which the particular address is allocated. TIPS memory allocation APIs internally use PMDK allocator. PMDK not only guarantees failure atomicity for memory allocation and free but also atomic persistence of a variable that stores the NVMM heap address [28]; TIPS passes an address in MLog to the PMDK memory allocation API that guarantees the address pointing to the allocated memory is persisted when returning from the API. Similarly, PMDK memory free guarantees atomic persistence for an address that is set to NULL. During recovery, all the non-NULL addresses with the “allocated” tag in MLog are deemed to be non-reachable. Such addresses are freed to avoid memory leaks as the insert operations that created these

addresses will be re-executed again from the OLog. Similarly, it is possible to re-execute the same OLog entry more than once; This can cause a double-free bug if a crash happens amidst a delete operation. To avoid this, TIPS logically removes the address from the plugged-in index, stores it in the MLog, and defers the actual memory free until the subsequent OLog reclamation. Because after reclaiming the OLog, the delete operation can not be re-executed again.

A running example. Suppose that inserting (or deleting) a key triggers split (or merge) on the leaf node A in a B+ tree, and a new leaf node A' is allocated (or the existing A freed). Say a crash happens before the completion of split (or merge) but after allocating A' (or freeing A). During the recovery, TIPS will re-execute the same insert (or delete) from the OLog, and it once again allocates a new leaf node A" (or free A again). This scenario leads to a persistent memory-leak of A' (or double-free of A). With the MLog, TIPS can reclaim the previously allocated node A' (or restore node A) during recovery to avoid persistent memory leak (or double-free).

4.3.2 UNDO Log (ULog)

What to log? ULog is used by the worker threads to guarantee failure-atomic updates to the plugged-in index. Generally, all the addresses that are being modified are required to be logged in the ULog. Instead, in TIPS, we leverage the OLog information to selectively log only the addresses that are needed for correct recovery. To decide whether to log a given address, the workers rely on two timestamp information: 1) the time at which the requested address is allocated (`alloc-ts`) and 2) the time of last OLog reclamation (`reclaim-ts`). If the requested address has its `alloc-ts` > `reclaim-ts` (*i.e.*, the address is allocated after the last OLog reclamation), then the operation to recreate the contents of this address is guaranteed to be present in the OLog. Hence the workers skip the UNDO logging. Otherwise, the workers first check if the requested address is already logged due to any previous write request. If so, the workers will skip the UNDO logging; else, they record the contents of the requested address in the ULog. The persist-thread defers the persistence of addresses in the ULog until the subsequent ULog reclamation. Thus, recurring updates to the same address can be batched and persisted at the start of every ULog reclamation.

A running example. Say a B+ tree node A is being modified 500 times between two ULog reclamations. Then a worker will log A in the ULog at the time of its first modification and reuse the same record until the next ULog reclamation. After the 500th update, say a ULog reclamation is triggered; the persist-thread before reclaiming the ULog will persist A with its latest update. If a crash happens before persisting A, during the recovery, TIPS correctly spots and reverts node A to the state before its first modification from the ULog. Then it re-executes all the 500 updates from the OLog to bring A to its latest state before the failure.

4.3.3 UNO Logging Reclamation

Log reclamation is triggered when any of OLog or ULog reaches their preset capacity threshold. The persist-thread always reclaims all the logs together even if only one of them reaches its capacity threshold. That is because, in TIPS, the information required for a correct recovery is distributed across all three logs. The reclamation yields for two cases; it waits until the (1) current epoch of background replay to end, and (2) pending scans to finish traversing the OLog. The UNO logging reclamation consists of the following two steps:

Step 1. Flush the addresses in the ULog and MLog. First, all the addresses in the ULog and MLog are persisted by calling `p-barrier`. This guarantees that all the writes that occurred since the last UNO reclamation are persisted.

Step 2. Reclaim the logs. Replayed OLog entries and persisted ULog entries in Step 1 are obsolete; they can be safely reclaimed to free up space for the incoming writes. Since the OLog has been reclaimed; we no longer required to keep track of the allocated and freed addresses; so the logically reclaimed addresses stored in the MLog are physically freed and then the MLog space can also be safely reclaimed.

Crash safety of reclamation. The reclamation procedure is crash-safe. The crash safety is guaranteed by *atomically setting the `flush_done` flag after the completion of Step 1*. Upon a crash, the recovery procedure first checks the `flush_done` flag. If the flag is set, it means that Step 1 has been completed successfully before the crash occurred and this guarantees that all the updates to the plugged-in index are persisted. Hence the recovery simply reclaims the remaining logs (Step 2) and terminates. If the flag is unset, then a standard recovery procedure described in the following section is followed.

4.4 Recovery

TIPS flushes all the logs and sets the tail pointer of the UNO log to NULL upon a safe termination. Therefore, if the tail is non-NULL upon a TIPS restart, it triggers the recovery procedure. If `flush_done` is set, recovery proceeds as described in the previous section. Otherwise, the recovery consists of three steps; (1) replay ULog to set the index to the exact consistent state that existed at the last UNO reclamation; (2) free all the newly allocated addresses (before the crash) from the MLog to prevent persistent memory leaks; (3) replay the OLog to get to the last successfully committed update before the crash. Note that the logs are replayed only up to their tail pointers to avoid executing any partial writes during the recovery.

If a crash occurs during the ULog replay (Step 1) or MLog free (Step 2), TIPS can continue from where it left off. This is because the changes to the plugged-in index due to ULog replay are immediately persisted. Also, freeing MLog after ULog replay does not affect the persistent state of the index. As described in §4.3.1, freeing MLog entries is guaranteed with atomic persistence to NULL, making this step failure-safe. On the other hand, if there is a crash while executing OLog

entries (Step 3), TIPS treats it similar to the failure during normal execution; *i.e.*, after rebooting, TIPS re-executes all the three steps. Because replaying OLog during recovery is treated similar to replaying the OLog in the background during normal execution. Note that re-executing OLogs after the ULog and MLog replay is idempotent, so it does not affect the consistency of the plugged-in index.

5 Correctness of TIPS

Theorem 1. TIPS guarantees Durable Linearizability (DL).

Proof. To guarantee DL, TIPS must satisfy three main invariants: (1) Effect of a committed operation can not be undone in the face of a crash. For all the non-commutative operations (*e.g.*, `insert(k1,v)`, `delete(k1)`), (2) the order of commit (*i.e.*, OLog write), and visibility (*i.e.*, DRAM-cache write) must always be maintained; and (3) also the background replay must be performed in the linearization order—in the same order as the operations are made visible in the TIPS frontend.

For (1), the effect of a write operation is visible only after updating the DRAM-cache entry, which is strictly done after persisting the OLog record. This guarantees that the readers will never observe the effects of non-durable write. For (2), the commit and the visibility order for non-commutative operations are synchronized using the per-bucket spinlock in the DRAM-cache as such operations is always guaranteed to happen on the same DRAM-cache bucket. A writer acquires the lock and commits its operation in the OLog with a timestamp (`commit-ts`). Then it adds the entry in the DRAM-cache and releases the lock, which guarantees that the order of commit and visibility is always the same for non-commutative operations. For (3), as described in §4.2.2, TIPS uses hashing to ensure non-commutative operations always go to the same worker queue. Then the worker sorts its queue in the `commit-ts` order and updates the plugged-in index to maintain the linearization order. For commuting operations, maintaining the linearization order is not required as they work on disjoint keys, so the effect of such operations will be the same regardless of the order in which they are executed.

By guaranteeing DL, TIPS eliminates the possibility of non-trivial crash consistency bugs as found in the previous work RECIPE [37]. Particularly, TIPS avoids the dirty read bugs as unpersisted writes are never visible to readers as guaranteed by (1). Even if certain reads are served from the DRAM-cache and say a crash happens, the OLog reply during the recovery will ensure that the plugged-in index is up to date with all the committed writes that happened before the crash. This ensures that all the pre-crash reads are still valid as they can be retrieved from the plugged-in index. □

Theorem 2. TIPS guarantees DL for scan operations.

Proof. A scan operation in TIPS traverses both the plugged-in index and the OLog, and then it merges the results. The DL guarantee can be violated if (1) the scan thread reads a

partially written OLog entry and (2) if it reads the unpersisted tail pointer. Both these cases can result in loss of data if a crash happens because of reading non-durable data. To avoid (1), the scan thread traverses the OLog from head to tail and this will hide any ongoing OLog writes as the failure atomicity of an OLog write is guaranteed by atomically updating the tail pointer (§4.1.5). To avoid (2), the scan thread before starting its traversal, checks if the tail pointer is persisted. If yes, it starts traversing; else, it backs off and retries. □

6 TIPS Implementation

TIPS is written in C, and the core library is about 5000 LoC. We use the hardware clock (`rdtscp` in x86 architecture) for scalable timestamp allocation. To address the clock skew between the CPU cores, we use ORDO [31] as done in many other previous works [33, 35]. Note that ORDO does not require any hardware extensions. We set the maximum DRAM-cache chain length to 5 beyond which the `gc`-thread starts reclaiming the applied entries in the chain. We chose this number after carefully considering the DRAM/NVMM random read performance ratio; random read latency in NVMM is about $5\times$ slower than DRAM [64] so that traversing more than 5 nodes in the collision chain do not pay off.

7 Evaluation

We evaluate TIPS by answering the following questions: (1) How do TIPS perform against prior conversion techniques (§7.2)? (2) How do the TIPS-indexes perform against prior NVMM-optimized indexes (§7.3)? (3) How do the TIPS-indexes scale for different workloads (§7.4)? (4) What is the sensitivity for DRAM-cache and UNO logging size (§7.5)? (5) How does TIPS impact real-world application (§7.6)?

Evaluation Platform. We use a system with Intel Optane DC Persistent Memory (DCPMM). It has two sockets with Intel Xeon Gold 5218 CPU with 16 cores per socket, 512GB of NVMM (4×128 GB) and 64 GB of DRAM (4×16 GB). We used GCC 8.3.1 with `-O3` flag to compile benchmarks and ran all our experiments on Linux kernel 4.18.16.

Configuration. We used YCSB [14]—a standard key-value store benchmark (Table 1) for all our evaluations. We used `index-microbench` [60] to generate the YCSB workloads. We ran the benchmarks for 32 million keys; we first populate an index with 32M keys and then run the workloads, which performs 32M operations. We use random integer and string keys with uniform distribution. We also evaluate the TIPS for large datasets and Zipfian distribution in §7.4. We preset the size of our per-thread OLog and the global ULog to 32MB each, the DRAM-cache to cache 25% of the total number of keys (300 MB) and the upper bound for the number of workers (W^{up}) to 32 (*i.e.*, half of the available CPUs). We present the sensitivity analysis for these configurations in §7.5. To ensure a fair comparison, we carefully chose the indexes as the existing conversion techniques are specific to certain concurrency

models. We also ported all the indexes to use the PMDK memory allocator. Porting all the RECIPE [37] and NVTraverse [21] indexes with PMDK allocator incurred about 1800 LoC. For all our evaluations (unless mentioned specifically), we use 32 threads to match the maximum physical CPU cores (without hyperthreads) available on our platform and present the scalability results for all the TIPS-indexes in §7.4.

7.1 Converting Volatile Index using TIPS

Converting an index using TIPS is simple (§3.2.6), and it requires only minimal LoC changes as shown in Table 2. For all the indexes, we replace the memory allocation (and free) with `tips_alloc` (and `tips_free`). Below we discuss how we annotated the NVMM stores with `tips_uolog_add`.

Index with Single Pointer Updates. Indexes for which write operations involve updating only a single pointer such as a Hash Table (HT), BST, and CLHT requires a `tips_uolog_add` before updating the HT/BST/CLHT node in the insert and delete logic, similar to the example shown in Figure 3. The lock-free indexes (LFHT/LFBST) use atomic Compare and Swap (CAS) and we added a `tips_uolog_add` before the CAS logic such that `tips_uolog_add` will be called again upon a CAS retry. Note that repeated UNDO logging will neither impact the correctness nor the performance as TIPS performs UNDO logging for an address only at the time of its first modification (§4.3.2). All such conversions require only 5-8 lines of change to the existing volatile codebase (Table 2).

Index with Multi-Pointer Updates. For the indexes like the B+tree or ART `tips_uolog_add` is added to backup the B+Tree/ART node before the triggering node split/merge operation. Also, `tips_uolog_add` is added before modifying the B+Tree/ART node in the normal case insert and delete logic. Totally it required only 11 and 9 LoC changes in the B+tree and ART codebase respectively. Note that none of the indexes require any modification in their read and scan logic.

Comparison with other Conversion Techniques. PRONTO [50] requires developers to manually add `op_begin()` and `op_commit()`. With NVTraverse [21] developers must first modify the index implementation to a "traversal" index and then manually add the `ensureReachable` and `makePersistent` APIs. Similar to the `tips_uolog_add` (§3.2.6), the aforementioned APIs will internally issue p-barrier without requiring any user intervention. Additionally, unlike PRONTO and NVTraverse, TIPS can be used on indexes supporting different concurrency models. Like TIPS, both PRONTO and NVTraverse formally prove that their conversion yields correct persistent algorithm by guaranteeing DL. As also reported in the NVTraverse paper,

YCSB Workload	Read-Write-Scan %	Workload Nature
A	50-50-0	Write Intensive
B	95-5-0	Read Intensive
C	100-0-0	Read Only
D	95-5-0	Read Latest
E	0-5-95	Short Range Scan

Table 1: Characteristics of YCSB workloads.

Indexes	Concurrency Control	LoC
Hash table (HT)	Readers-writer lock	5/211
Lock-Free HT (LFHT) [51]	Non-blocking reads and writes	5/199
Binary Search Tree (BST)	Readers-writer lock	5/203
Lock-Free BST (LFBST) [12]	Non-blocking reads and writes	5/194
B+tree	Readers-Writer lock	8/711
Adaptive Radix Tree (ART) [40]	Non-blocking reads/blocking writes	9/1.5K
Cache-line Hash Table (CLHT) [16]	Non-blocking reads/blocking writes	8/2.8K
Redis [8]	Blocking reads and writes	18/10K

Table 2: Lines of code (LoC) to convert volatile indexes using TIPS.

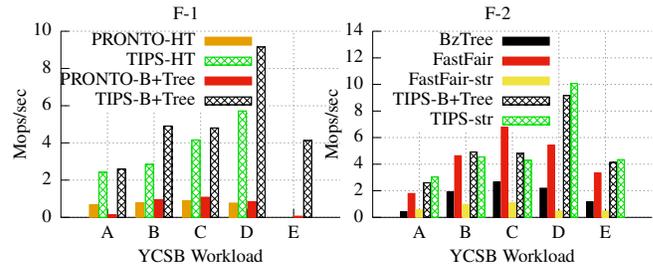


Figure 4: Performance comparison of TIPS against PRONTO for Hash Table (HT) and B+Tree (F-1) and TIPS-B+Tree against the NVMM-optimized B+Tree indexes— FastFair and BzTree (F-2).

RECIPE [37] can not always guarantee a correct conversion, even for the indexes that fall under their prescribed condition.

Moreover, RECIPE does not formalize the guarantees of their conversions and does not discuss the implications of guaranteeing BDL. Their updated ArXiv version [38] prescribes to selectively add p-barrier to specific loads to guarantee DL. But it is left to the developers to figure out the loads that needs to be correctly flushed; this further complicates the conversion. Alternatively, TIPS requires only minimal and simple modifications in the volatile codebase. We also formally describe how our conversion guarantees DL and yields a correct persistent algorithm for all our conversions.

7.2 TIPS vs. Other Conversion Techniques

TIPS vs. PRONTO [50]. PRONTO is the state-of-the-art technique to convert globally blocking indexes with DL guarantee. As shown in Figure 4 (F1), both TIPS-HT and TIPS-B+Tree outperform the PRONTO counterparts by 20× across all workloads. Although both TIPS (TIPS-HT, TIPS-B+Tree) and PRONTO use RW lock for concurrency, TIPS can process the reads and writes concurrently in the DRAM-cache, and hence it shows a better performance. Also, PRONTO's overhead mainly comes from the synchronous waiting of writers for its background thread to complete the logging. Our performance profiling on the PRONTO-HT reveals that about 25% of the execution time is spent on synchronous waiting. In TIPS, there is no such synchronous waiting as it does not have any separate logging threads. Instead, writers will perform logging in their private OLog. Another source of overhead is the blocking during snapshots, which accounts for 8% of the execution time. Moreover, PRONTO builds its index on DRAM, so it can not scale beyond the DRAM capacity while TIPS can scale up to the NVMM capacity. This is a critical design benefit because legacy applications adopt NVMM not

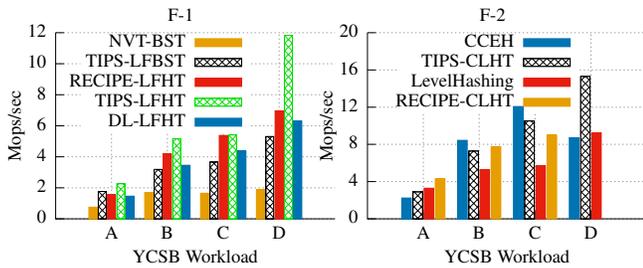


Figure 5: Performance comparison of TIPS with NVTraverse (F-1), RECIPE (F-1, F-2) and TIPS-CLHT with NVMM-optimized hash indexes—CCEH and LevelHashing (F-2).

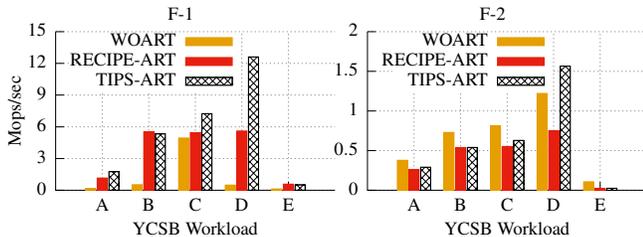


Figure 6: Performance comparison of TIPS-ART with RECIPE-ART and WOART for 32 threads (F-1) and 1 thread (F-2).

just for durability but also for its large in-memory capacity.

TIPS vs. NVTraverse [21]. NVTraverse is the state-of-the-art technique to convert lock-free indexes with DL guarantee. As shown in Figure 5 (F-1), TIPS-LFBST outperforms NVT-BST by up to $3\times$ across all workloads. Further analysis revealed that on average, each read and write in NVT-BST incurs 6 and 17 p-barriers, respectively, in the critical path. While TIPS-LFBST incurs only 2 p-barriers for each write in the critical path and reads never require p-barrier, thanks to the UNO logging and the DRAM-cache. Moreover, TIPS serves up to 25% of read requests from the DRAM-cache. So the readers do not need to traverse the BST on the NVMM for 25% of its read requests and hence a better performance.

TIPS vs. RECIPE [37]. Comparing TIPS and RECIPE in terms of performance is not an apple-to-apple comparison as RECIPE supports only a weaker consistency (*i.e.*, BDL). Besides performance, we also stress how hard it is to achieve DL without trading off performance. Figure 5 and Figure 6 compare the performance of TIPS and RECIPE for LFHT, ART [39], and CLHT [16], respectively. TIPS indexes perform similar or better than RECIPE indexes across all workloads except for CLHT in workload A. This is because writes to CLHT incurs only one cacheline modification and one p-barrier in RECIPE. While in TIPS-CLHT, it incurs two p-barriers to commit the OLog. Nonetheless, TIPS-CLHT supports DL, and it performs mostly similar to NVMM-optimized hash indexes CCEH [53] and LevelHashing [66]. An easy way to guarantee DL, as proposed by Izraelevitz *et al.* [29] is to add a p-barrier for every reads and writes. We followed it to make a DL version of the RECIPE hash table (DL-LFHT in Figure 5). Such a conversion leads up to $1.4\times$ drop in performance. One can also perform index-specific optimizations like the NVTraverse to guarantee DL. However, it requires

expertise in NVMM programming and in-depth knowledge of the volatile index. Conversely, TIPS achieves DL with good performance and, notably, in an index-agnostic way.

7.3 TIPS vs. NVMM-optimized Indexes

TIPS-B+tree. Figure 4 (F-2) shows the performance of TIPS-B+tree against FastFair [26], and BzTree [11]. TIPS outperforms BzTree by up to $3\times$ across all workloads; BzTree uses CoW and PMwCAS [59] to support crash consistency, and this generates a lot of NVMM write traffic. Unlike BzTree, TIPS-B+tree supports low overhead crash consistency using UNO logging and hence a better performance. TIPS performs similar or better than FastFair except for workload C. FastFair, with its smaller fanout (16), provides good point query performance, and TIPS-B+tree with a larger fanout (128) provides a good range query performance than FastFair. For string keys, FastFair (FastFair-str) performs up to $5\times$ slower than TIPS (TIPS-str) as it loses its cache efficiency due to additional pointer chasing to retrieve string keys. Note that TIPS stores pointer to its keys for both string and integer keys; therefore no significant performance drop is observed.

TIPS-ART. In Figure 6, we present the performance of TIPS-ART and WOART [36]—volatile ART variant designed for NVMM. WOART is single-threaded, and hence we used a global lock for concurrency as done in previous work [37, 38]. WOART performs up to $40\times$ slower than TIPS-ART across different workloads due to its poor concurrency model. For a single thread, TIPS-ART performs similar to WOART except for workload E. For workload E, the performance of TIPS-ART and RECIPE-ART are almost identical; this proves our claim in §4.1.6 that the additional OLog traversal in scan operations imposes negligible overhead. Our performance profiling revealed that only 2-3% of the time is spent on traversing OLog regardless of thread count.

7.4 Analysis on TIPS Design

Write Scalability. Figure 7 shows the scalability of the TIPS indexes. For write-intensive workload A, all TIPS indexes show good performance and scalability; for TIPS-B+Tree a sharp increase is observed after 16 threads. Because (1) for lower thread counts, the aggregate OLog size is less (OLog is per-thread). Moreover, as TIPS-B+Tree uses global RW lock TIPS backend writes are slower than the concurrent frontend writes, and consequently, foreground writers are blocked due to the lack of OLog space. (2) For higher thread counts, foreground blocking time is significantly reduced with a larger aggregate OLog size. Although the TIPS-HT uses RW lock, it is per-bucket and hence a better level of concurrency than the TIPS-B+tree. This enables TIPS backend to reply the OLog concurrently and hence a better performance for TIPS-HT. Still, for 16 threads, TIPS-B+tree outperforms PRONTO-B+Tree which also uses global RW lock by up to $3\times$.

Read Scalability. All TIPS indexes show good scalability for read-intensive workloads. Indexes supporting concurrent

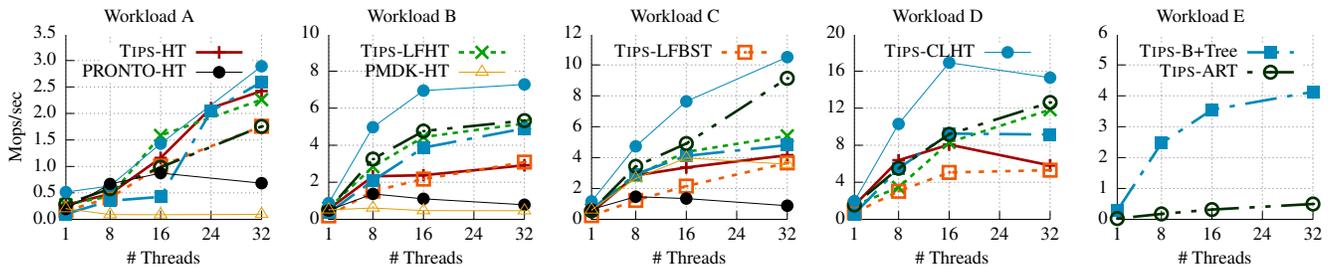


Figure 7: Scalability of TIPS-HT (RW lock), TIPS-B+Tree (RW lock), TIPS-LFHT, TIPS-LFBST, TIPS-CLHT and TIPS-ART.

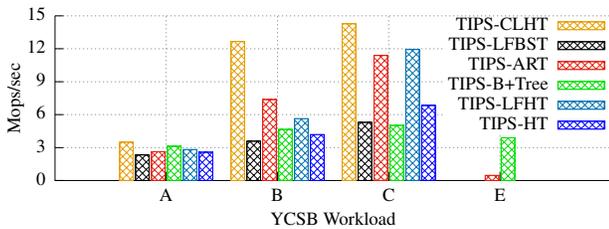


Figure 8: Performance of TIPS indexes for Zipfian workloads.

reads show good performance; for instance, TIPS-LFHT performs up to $1.2\times$ better than TIPS-HT (RW lock) in workload C. All indexes regardless of their concurrency model shows high performance for workload D. Because workload D follows read-latest distribution, latest writes are being repeatedly read. Fortunately, the latest writes are most likely to be present in the DRAM-cache; hence a higher read hit ratio and consequently better performance. Although we cache only 25% of the keys for all workloads, the read hit for workload D is about 70%, while for other workloads it is less than 25%.

Impact of UNO Logging. Both TIPS-HT and PMDK-HT (Figure 7) use RW lock, and they both use the UNDO logging to guarantee crash consistency while updating the hash table. However, despite the similarities, TIPS-HT significantly outperforms PMDK-HT. The main performance bottleneck in PMDK is the logging operations in the critical path. This is evident from Figure 7, PMDK-HT performs and scales on par with TIPS-HT for workload C (100% reads), but its performance plateaus even for a small fraction of writes (5%) in workload B. With UNO logging, UNDO logging is kept off the critical path and consequently making TIPS scale better.

Impact of Skewed Workloads. Figure 8 shows the performance of TIPS indexes for Zipfian distribution; all indexes shows up to $2\times$ better performance than the uniform distribution. Particularly for workload A, the chances of write coalescing in the ULog increases due to frequent updates to the same address. This reduces the number of UNDO logging performed, and further analysis revealed that the number of UNDO logging performed is reduced by 16% than that of uniform distribution. This further accelerates the background writes, and thus overall write performance is improved. For read-intensive workloads, repeated reading of hotkeys results in more DRAM-cache hits and eventually a better read performance. On average, the DRAM-cache hit ratio is increased by 5% for the Zipfian distribution. Overall, TIPS, in partic-

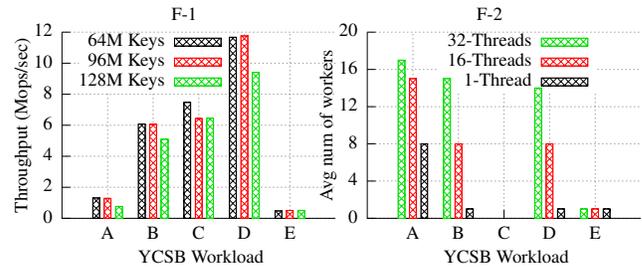


Figure 9: Studying the impact of large dataset (F-1) for 32 threads and adaptive scaling (F-2) for varying threads with TIPS-ART.

ular, UNO logging and DRAM-cache are well equipped to handle the skewed workloads. Note that we did not evaluate workload D as it supports read-latest distribution by default.

Impact of Large Dataset. Figure 9 (F-1) presents the performance of TIPS-ART for large datasets. While the performance for 64M and 96M keys is mostly the same across all the workloads, there is up to a 23% drop in performance for 128M keys. Particularly for workload C, the performance drop is also observed for 96M keys; for a larger dataset, the ART index grows bigger, and it results in increased pointer chasing to get to the leaf nodes, and hence there is a slight dip in the performance. Note that the RECIPE-ART also exhibits a performance drop of up to 16% across all workloads for 128M keys. Overall, this evaluation shows that TIPS as a system can handle much larger datasets effectively.

Impact of Adaptive Scaling. Figure 9 (F-2) shows the average number of background workers used for TIPS-ART. More workers are used for workload A as it is write-intensive, for all the other workloads workers count is relatively less as the write ratio is marginal. No workers are created for workload C as it is read-only. Workload E has the same write ratio (5%) as B and D, but TIPS employs only one worker for E. Because ART's scan is inherently slower, hence the foreground threads spend most of the time on the scan operation (*i.e.*, foreground writes are slow), this gives TIPS enough time propagate the updates. Thus our adaptive scaling can effectively adapt for the nature of workloads and the plugged-in index.

7.5 Sensitivity Analysis

Sensitivity to DRAM-cache Size. As shown in Figure 10, for read-intensive workloads, about $1.8\text{-}2.8\times$ performance increase is observed as % keys cached increases. This is because the read hit ratio increases as more keys are being

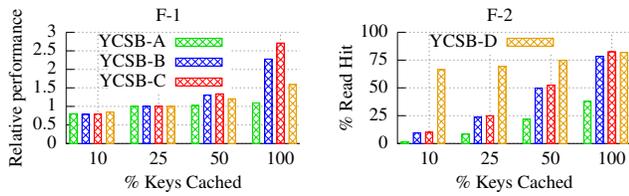


Figure 10: Performance sensitivity of TIPS-B+tree (F-1) and read hit % (F-2) for the varying DRAM-cache size. X-axis represents the % of keys cached in the DRAM-cache (default = 25%).

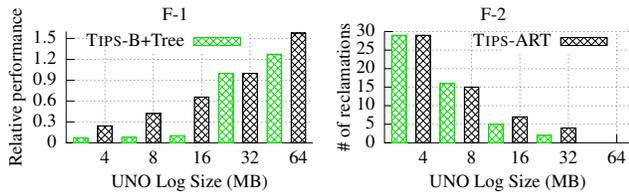


Figure 11: Performance sensitivity (F-1) and the number of log reclamations triggered (F-2) in TIPS-B+Tree and TIPS-ART for the varying UNO log size for Workload A (default = 32MB).

cached, enabling readers to complete their reads on the faster DRAM. Since workload A is write-intensive, it is less sensitive to DRAM-cache size. As more writes happen on the DRAM-cache, the applied keys are actively evicted, and it stores mostly the newly written keys. This poorly impacts the read hit ratio, and consequently, readers are forced to fall back to the B+tree on the NVMM.

Sensitivity to UNO Log Size. Figure 11 illustrates the performance of TIPS-ART, TIPS-B+tree for different UNO log sizes for the write-heavy workload A. As shown, there is a $4\times$ and $9\times$ performance drop for TIPS-ART and TIPS-B+tree with 32MB (default) and 4MB log size, respectively. This is because as the log size becomes smaller, the foreground writers are blocked for more time as TIPS-B+tree (RW lock) has a single-threaded backend. Whereas TIPS-ART supports concurrent backend and is relatively less affected by decreasing log size. Both TIPS-ART and TIPS-B+tree shows a $1.6\times$ performance increase for 64MB because the log size is big enough to completely buffer all writes, and zero reclamations are triggered for 64MB. Also, note that more than 3 reclamations are triggered for a default log size of 32MB. The impact of smaller log sizes is relatively marginal for read-heavy workloads. The performance change is negligible until 8MB, and about a 20% performance drop is observed for 4MB log size.

7.6 Real-world Application: Redis

We ported a popular DRAM key-value store, Redis [8], using TIPS (TIPS-Redis). We compare its performance with the vanilla Redis running on the DRAM (DRAM-Redis) and NVMM (NVMM-Redis), and also with Intel’s PMEM-Redis [7]. Note that NVMM-Redis does not ensure crash consistency and PMEM-Redis stores only the values in NVMM. Figure 12 shows the performance of Redis GET and SET operation evaluated using Redis-Benchmark [9]. We ran the benchmark for 32M keys (8-bytes) with a uniform random

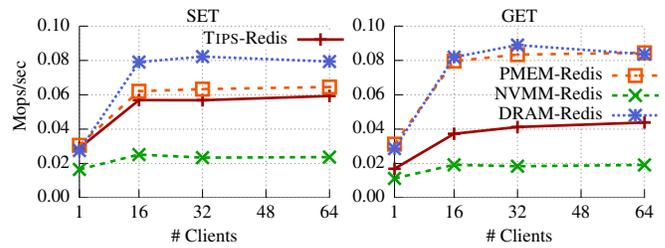


Figure 12: Performance comparison of TIPS-Redis with vanilla Redis running on DRAM and NVMM, and Intel’s PMEM-Redis.

distribution. For SET operations, TIPS-Redis consistently outperforms the NVMM-Redis by $2.5\times$, and it performs up to $1.5\times$ and $1.1\times$ slower than the DRAM-Redis and PMEM-Redis, respectively. For the GET operations, TIPS-Redis perform up to $2\times$ better than the NVMM-Redis and up to $2.2\times$ slower than the DRAM-Redis and PMEM-Redis. TIPS-Redis maintains all the data and the Redis core on the NVMM, so (1) it provides a larger in-memory capacity ($4\times$ in our experiment) and immediate durability. (2) Both PMEM-Redis and DRAM-Redis take about 100 seconds to restore data from disk every time a server instance is created. While TIPS-Redis takes less than 1 second to recover upon safe termination.

7.7 Recovery

We performed the recovery test on all the TIPS-indexes. We injected crash 200 times arbitrarily using SIGKILL, similar to previous work [37, 41]. We also tested a crash during the recovery procedure. All TIPS-indexes successfully recovered after every crash. The worst-case recovery time would be a crash happening when the OLog is full. To measure this time, we injected a crash just when the OLog becomes full. Recovery time ranges between 0.5 and 9 seconds depending on the number of OLogs and concurrency control of the index.

8 Conclusion

We propose TIPS, a framework to systematically make volatile indexes and in-memory key-value stores persistent. At its core, TIPS adopts a novel DRAM-NVMM tiering to support index-agnostic conversion and durable linearizability. With the tiered concurrency model, TIPS achieves good scalability, performance, and enhanced applicability. UNO logging protocol is critical to achieve low crash consistency overhead and prevent persistent memory leaks. In our evaluation, we showed that TIPS could be effectively applied to indexes with varying concurrency models and the TIPS-enabled indexes shows excellent performance against the state-of-the-art index conversion techniques and NVMM-optimized indexes.

Acknowledgement

We thank the anonymous reviewers and Haris Volos (our shepherd) for their insightful comments and feedback. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2014-3-00035).

References

- [1] Accelerating Redis with Intel Optane DC Persistent Memory. https://ci.spdk.io/download/2019-summit-prc/02_Presentation_13_Accelerating_Redis_with_Intel_Optane_DC_Persistent_Memory_Dennis.pdf.
- [2] Aerospike Performance on Intel Optane Persistent Memory. <https://www.aerospike.com/blog/performance-on-intel-optane-persistent-memory/>.
- [3] Bringing The Latest Persistent Memory Technology to Redis Enterprise. <https://redislabs.com/blog/persistent-memory-and-redis-enterprise/>.
- [4] Intel Optane DC Persistent Memory NoSQL Performance Review. <https://www.storagereview.com/review/intel-optane-dc-persistent-memory-nosql-performance-review>.
- [5] Key/Value Datastore for Persistent Memory. <https://github.com/pmem/pmemkv>.
- [6] Optimize Redis With Next Gen NVM. https://www.snia.org/sites/default/files/SDC/2018/presentations/PM/Shu_Kevin_Optimize_Redis_with_NextGen_NVM.pdf.
- [7] Pmem-Redis. <https://github.com/pmem/pmem-redis/>.
- [8] Redis. <https://github.com/antirez/redis>.
- [9] Redis Benchmark - How fast is Redis? <https://redis.io/topics/benchmarks>.
- [10] Anandtech. Intel Launches Optane DIMMs Up To 512GB: Apache Pass Is Here!, 2018. URL: <https://www.anandtech.com/show/12828/intel-launches-optane-dimms-up-to-512gb-apache-pass-is-here>.
- [11] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A High-performance Latch-free Range Index for Non-volatile Memory. In *Proceedings of the 44th International Conference on Very Large Data Bases (VLDB)*, Rio De Janerio, Brazil, August 2018.
- [12] Bapi Chatterjee, Nhan Nguyen, and Philippas Tsigas. Efficient Lock-Free Binary Search Trees. In *Proceedings of the 33th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, Paris, France, July 2014.
- [13] Shimin Chen and Qin Jin. Persistent B+-trees in Non-volatile Main Memory. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*, Hawaii, USA, September 2015.
- [14] 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)*, pages 143–154, Indianapolis, Indiana, USA, June 2010. ACM.
- [15] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.
- [16] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, March 2015.
- [17] Anthony Demeri, Wook-Hee Kim, R. Madhava Krishnan, Jaeho Kim, Mohannad Ismail, and Changwoo Min. Poseidon: Safe, fast and scalable persistent memory allocator. In *Proceedings of the 21st*, Delft, Netherlands, December 2020.
- [18] 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 the 2013 ACM SIGMOD/PODS Conference*, pages 1243–1254, New York, USA, June 2013. ACM.
- [19] Franz Färber, Sang Kyun Cha, Jürgen Primisch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. Sap hana database: Data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, January 2012. URL: <http://doi.acm.org/10.1145/2094114.2094126>, doi:10.1145/2094114.2094126.
- [20] Keir Fraser. Practical Lock Freedom, 2004. <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf>.
- [21] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey. In *Proceedings of the 2020 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, London, UK, June 2020.

- [22] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Changwoo Min, and Dongyoon Lee. Witcher : Detecting crash consistency bugs in non-volatile memory programs, 2020. [arXiv:2012.06086](https://arxiv.org/abs/2012.06086).
- [23] Swapnil Haria, Mark D. Hill, and Michael M. Swift. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, March 2020.
- [24] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of Memory Reclamation for Lockless Synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, 2007.
- [25] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
- [26] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-addressable Persistent B+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, pages 187–200, Oakland, California, USA, February 2018.
- [27] Intel. C++ bindings for libpmemobj (part 6) - transactions, 2016. URL: <http://pmem.io/2016/05/25/cpp-07.html>.
- [28] INTEL. PMDK man page: pmemobj_alloc, 2019. URL: http://pmem.io/pmdk/manpages/linux/v1.5/libpmemobj/pmemobj_alloc.3.
- [29] Joseph Izraelevitz, Hammurabi Mendes, and Michael Scott. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Proceedings of the 30th International Conference on Distributed Computing (DISC)*, Paris, France, September 2016.
- [30] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NovelLSM. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.
- [31] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. A Scalable Ordering Primitive for Multicore Machines. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*, pages 34:1–34:15, Porto, Portugal, April 2018. ACM.
- [32] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE)*, pages 195–206, Hannover, Germany, April 2011.
- [33] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. Mvrlu: Scaling read-log-update with multi-versioning. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 779–792, Providence, RI, April 2019. ACM.
- [34] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 468–479, Davis, CA, USA, December 2013.
- [35] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable Transactional Memory Can Scale with Timestone. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, March 2020.
- [36] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, February–March 2017.
- [37] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.
- [38] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. October 2019. [arXiv:1909.13670v2](https://arxiv.org/abs/1909.13670v2) [cs.DC], <https://arxiv.org/abs/1909.13670v2>. [arXiv:arXiv:1909.13670v2](https://arxiv.org/abs/1909.13670v2).
- [39] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*, pages 38–49, Brisbane, Australia, April 2013.
- [40] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of Practical Synchronization. In *Proceedings of the International Workshop on*

Data Management on New Hardware, pages 3:1–3:8, San Francisco, California, June 2016.

- [41] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating Persistent Memory Range Indexes. In *Proceedings of the 45th International Conference on Very Large Data Bases (VLDB)*, Los Angeles, CA, August 2019.
- [42] Jihang Liu, Shimin Chen, and Lujun Wang. LB+Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, August 2020.
- [43] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, August 2020.
- [44] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. ROART: Range-query optimized persistent ART. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–16, Santa Clara, CA, February 2021.
- [45] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys)*, pages 183–196, Bern, Switzerland, April 2012.
- [46] Virendra J. Marathe, Margo Seltzer, Steve Blyan, and Tim Harris. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *Proceedings of the 17th Workshop on Hot Topics in Storage and File Systems*, Santa Clara, CA, July 2017.
- [47] Ajit Mathew and Changwoo Min. HydraList: A Scalable In-Memory Index Using Asynchronous Updates and Partial Replication. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, August 2020.
- [48] Paul E. McKenney. Structured deferral: Synchronization via procrastination. *ACM Queue*, pages 20:20–20:39, 1998.
- [49] A. Memaripour and S. Swanson. Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software. In *Proceedings of the 36th International Conference on Computer Design*, Hartford, CT, October 2018.
- [50] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, March 2020.
- [51] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, page 73–82, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/564870.564881.
- [52] Micro. 3D XPoint Technology, 2019. URL: <https://www.micron.com/products/advanced-solutions/3d-xpoint-technology>.
- [53] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*, Boston, MA, February 2019.
- [54] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dali: A Periodically Persistent Hash Map. In *Proceedings of the 31st International Conference on Distributed Computing (DISC)*, Vienna, Austria, October 2017.
- [55] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2015 ACM SIGMOD/PODS Conference*, San Francisco, CA, USA, June 2016.
- [56] Ismail Oukid and Wolfgang Lehner. Data structure engineering for byte-addressable non-volatile memory. In *Proceedings of the 2017 ACM SIGMOD/PODS Conference*, Chicago, Illinois, USA, May 2017.
- [57] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–32, Farmington, PA, November 2013. ACM.
- [58] Li Wang, Zining Zhang, Bingsheng He, and Zhenjie Zhang. PA-Tree: Polled-Mode Asynchronous B+ Tree for NVMe. In *Proceedings of the 36th IEEE International Conference on Data Engineering (ICDE)*, Dallas, TX, April 2020.
- [59] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy Lock-Free Indexing in Non-Volatile Memory.

- In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE)*, pages 461–472, Paris, France, April 2018.
- [60] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 ACM SIGMOD/PODS Conference*, pages 473–488, Houston, TX, USA, June 2018.
- [61] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. Interval-Based Memory Reclamation. In *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, wien, Austria, March 2018.
- [62] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. PMThreads: Persistent Memory Threads Harnessing Versioned Shadow Copies. In *Proceedings of the 2020 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, London, UK, June 2020.
- [63] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.
- [64] 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)*, Santa Clara, CA, February 2020.
- [65] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, February 2015.
- [66] Pengfei Zuo, Yu Hua, and Jie Wu. 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)*, Carlsbad, CA, October 2018.

Improving Performance of Flash Based Key-Value Stores Using Storage Class Memory as a Volatile Memory Extension

Hiwot Tadese Kassa
University of Michigan

Jason Akers
Facebook, Inc.

Mrinmoy Ghosh
Facebook, Inc.

Zhichao Cao
Facebook, Inc.

Vaibhav Gogte
University of Michigan

Ronald Dreslinski
University of Michigan

Abstract

High-performance flash-based key-value stores in data-centers utilize large amounts of DRAM to cache hot data. However, motivated by the high cost and power consumption of DRAM, server designs with lower DRAM per compute ratios are becoming popular. These low-cost servers enable scale-out services by reducing server workload densities. This results in improvements to overall service reliability, leading to a decrease in the total cost of ownership (TCO) for scalable workloads. Nevertheless, for key-value stores with large memory footprints these reduced DRAM servers degrade performance due to an increase in both IO utilization and data access latency. In this scenario a standard practice to improve performance for sharded databases is to reduce the number of shards per machine, which degrades the TCO benefits of reduced DRAM low-cost servers. In this work, we explore a practical solution to improve performance and reduce costs of key-value stores running on DRAM constrained servers by using Storage Class Memories (SCM).

SCM in a DIMM form factor, although slower than DRAM, are sufficiently faster than flash when serving as a large extension to DRAM. In this paper, we use Intel® Optane™ PMem 100 Series SCMs (DCPMM) in AppDirect mode to extend the available memory of RocksDB, one of the largest key-value stores at Facebook. We first designed hybrid cache in RocksDB to harness both DRAM and SCM hierarchically. We then characterized the performance of the hybrid cache for 3 of the largest RocksDB use cases at Facebook (WhatsApp, Tectonic Metadata, and Laser). Our results demonstrate that we can achieve up to 80% improvement in throughput and 20% improvement in P95 latency over the existing small DRAM single-socket platform, while maintaining a 43-48% cost improvement over the large DRAM dual socket platform. To the best of our knowledge, this is the first study of the DCPMM platform in a commercial data center.

1 Introduction

High-performance storage servers at Facebook come in two flavors. The first, *2P server*, has two sockets of compute and a large DRAM capacity as shown in Figure 1a and provides

excellent performance at the expense of high power and cost. In contrast, *1P server* (Figure 1b), has one socket of compute and the DRAM-to-compute ratio is half of the *2P server*. The advantages of *1P server* are reduced cost, power, and increased rack density [1]. For services with a small DRAM footprint, *1P server* is the obvious choice. A large number of services in Facebook fit in this category.

However, a class of workloads that may not perform adequately on a reduced DRAM server and take advantage of the cost benefits of *1P server* at Facebook are flash-based key-value stores. Many of these workloads use RocksDB [2] as their underlying storage engine. RocksDB utilizes DRAM for caching frequently referenced data for faster access. A low DRAM to storage capacity ratio for these workloads will lead to high DRAM cache misses, resulting in increased flash IO pressure, longer data access latency, and reduced overall application throughput. Flash-based key-value stores in Facebook are organized into shards. An approach to improve the performance of each shard on DRAM constrained servers is to reduce the number of shards per server. However, this approach can lead to an increase in the total number of servers required, lower storage utilization per server, and dilutes the TCO benefits of the *1P server*. This leaves us with the difficult decision between *1P server*, which is cost-effective while sacrificing performance, or *2P server* with great performance at high cost and power. An alternative solution that we explore in this paper is utilizing recent Intel® Optane™ PMem 100 Series SCMs (DCPMM) [3] to efficiently expand the volatile memory capacity for *1P server* platforms. We use SCM to build new variants of the *1P server* platforms as shown in Figure 1c. In *1P server variants*, the memory capacity of *1P server* is extended by providing large SCM DIMMs alongside DRAM on the same DDR and bus attached to the CPU memory controller.

Storage Class Memory (SCM) is a technology with the properties of both DRAM and storage. SCMs in DIMM form factor have been studied extensively in the past because of their attractive benefits including byte-addressability, data persistence, cheaper cost/GB than DRAM, high density, and their relatively low power consumption. This led to abundant research focusing on the use cases of SCM as memory

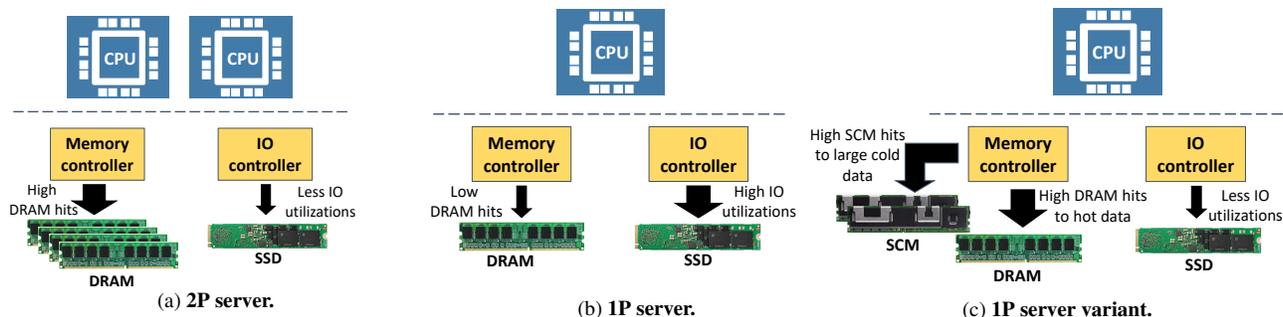


Figure 1: Server configurations with different DRAM sizes: (a) Server with 256GB memory, high hit rate to DRAM and low IO utilization (b) Server with reduced (64GB) memory, lower DRAM hit rate and increased in IO utilization and (c) Server with reduced memory and SCM added to the memory controller, high hit rate to DRAM & SCM due to optimized data placement, which decreases IO utilization.

Table 1: Example of memory characteristics of DRAM, SCM and flash per module taken from product specifications.

Characteristics	DRAM	SCM	Flash
Idle read latency (ns)	75	170	85,000
Read bandwidth (GB/s)	15	2.4	1.6
Power (mW / GB)	375	98	5.7
DRAM Relative Cost per GB	1	0.38	0.017
Granularity	byte addressable	byte addressable	block based
Device Capacity (GB)	32	128	2048

and persistent storage. The works range from optimizations with varying memory hierarchy configurations [4–8], novel programming models and libraries [9–11], and file system designs [12–14] to adopt this emerging technology. Past research was focused primarily on theoretical or simulated systems, but the recent release of DCPMM-enabled platforms from Intel motivates studies based on production-ready platforms [15–24]. The memory characteristics of DRAM, DCPMM, and flash are shown in Table 1. Even though DCPMM has higher access latency and lower bandwidth than DRAM it has a much larger density and lower cost, and its access latency is two orders of magnitude lower than flash. Currently, DCPMM modules come in 128GB, 256GB, and 512GB capacities, much larger than DRAM that typically ranges from 4GB to 32GB in a data-center environment. Hence we can get a tremendously larger density with DCPMM. If we efficiently (cost and performance) use this memory as an extension to DRAM, this would enable us to build dense, flexible, servers with large memory and storage, while using fewer DIMMs and lowering the total cost of ownership (TCO).

Although recent works demonstrated the characteristics of SCM [15, 18], the performance gain achieved in large commercial data-centers by utilizing SCM remains unanswered. There are open questions on how to efficiently configure DRAM and SCM to benefit large scale service deployments in terms of cost/performance. Discovering the use cases within a large scale deployment that profit from SCM has also been challenging. To address these challenges for RocksDB, we first profiled all flashed-based KV store deployments at Facebook to identify where SCM fits in our environment. These studies revealed that we have abundant

read-dominated workloads, which focused our design efforts on better read performance. This has also been established in previous work [25–27] where faster reads improved overall performance for workloads serving billions of reads every second. Then, we identified the largest memory consuming component of RocksDB, the block cache used for serving read requests, and redesigned it to implement a hybrid tiered cache that leverages the latency difference of DRAM and SCM. In the hybrid cache, DRAM serves as the first tier cache accommodating frequently accessed data for fastest read access, while SCM serves as a large second tier cache to store less frequently accessed data. Then, we implemented cache admission and memory allocation policies that manage the data transfer between DRAM and SCM. To evaluate the tiered cache implementations we characterize three large production RocksDB use cases at Facebook using the methods described in [28] and distilled the data into new benchmark profiles for db_bench [29]. Our results show that we can achieve 80% improvement to throughput, 20% improvement in P95 latency, and 43-48% reduction in cost for these workloads when we add SCM to existing server configurations. In summary, we make the following contributions:

- We characterized real production workloads, identified the most benefiting SCM use case in our environment, and developed new db_bench profiles for accurately benchmarking RocksDB performance improvement.
- We designed and implemented a new hybrid tiered cache module in RocksDB that can manage DRAM and SCM based caches hierarchically, based on the characteristics of these memories. We implemented three admission policies for handling data transfer between DRAM and SCM cache to efficiently utilize both memories. This implementation will enable any application that uses RocksDB as its KV Store back-end to be able to easily use DCPMM.
- We evaluated our cache implementations on a newly released DCPMM platform, using commercial data center workloads. We compared different DRAM/SCM size server configurations and determined the cost and per-

formance of each configuration compared to existing production platforms.

- We were able to match the performance of large DRAM footprint servers using small DRAM and additional SCM while decreasing the TCO of read dominated services in production environment.

The rest of the paper proceeds as follows. In Section 2 we provide a background of RocksDB, the DCPMM hardware platforms, and brief description of our workloads. Sections 3 and 4 explain the designs and implementation of the hybrid cache we developed. In Section 5 we explain the configurations of our systems and the experimental setup. Our experimental evaluations and results are provided in Section 6. We then discuss future directions and related works in Section 7 and Section 8 respectively, and conclude in Section 9.

2 Background

2.1 RocksDB architecture

A Key-value database is a storage mechanism that uses key-value pairs to store data where the key uniquely identifies values stored in the database. The high performance and scalability of key-value databases promote their widespread use in large data centers [2, 30–32]. RocksDB is a log-structured-merge [33] key-value store engine developed based on the implementation of LevelDB [30]. RocksDB is an industry standard for high performance key-value stores [34]. At Facebook RocksDB is used as the storage engine for several data storage services.

2.1.1 RocksDB components and memory usage

RocksDB stores key-value pairs in a Sorted String Table (SST) format. Adjacent key-value data in SST files are partitioned into data blocks. Other than the data block, each SST file contains Index and Filter blocks that help to facilitate efficient lookups in the database. SST files are organized in levels, for example, Level0 - LevelN, where each level comprises multiple SST files. Write operations in RocksDB first go to an in-memory write buffer residing in DRAM called the memtable. When the buffered data size in the memtable reaches a preset size limit RocksDB flushes recent writes to SST files in the lowest level (Level0). Similarly, when Level0 exhausts its size limit, its SST files are merged with SST files with overlapping key-values in the next level and so on. This process is called compaction. Data blocks, and optionally, index and filter blocks are cached (typically uncompressed) in an in-memory component called the Block Cache that serves read requests from RocksDB. The size of the Block Cache is managed by global RocksDB parameters. Reads from the database are attempted to be serviced first from the memtable, then next from the Block Cache(s) in DRAM, and finally from the SST files if the key is not found in memory. Further details about Rocksdb are found in [2]. The largest use of memory in RocksDB comes from the Block Cache, used for reads.

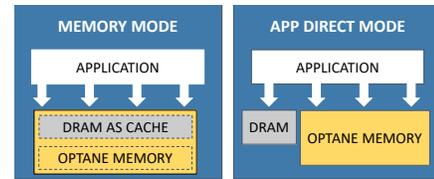


Figure 2: Intel® Optane™ memory operation modes overview.

Therefore, in this work we optimize the RocksDB using SCM as volatile memory for the Block Cache.

2.1.2 Benchmarking RocksDB

One of the main tools to benchmark RocksDB is `db_bench` [29]. `Db_bench` allows us to mock production RocksDB runs by providing features such as multiple databases, multiple readers, and different key-value distributions. Recent work [28] has shown how we can create realistic `db_bench` workloads from production workloads. To create evaluation benchmarks for SCM we followed the procedures given in [28, 35].

2.2 Intel® Optane™ DC Persistent Memory

Intel® Optane™ DC Persistent Memory based on 3D XPoint technology [36, 37], is the first commercially available non-volatile memory in the DIMM form factor and resides on the same DDR bus as DRAM [3]. DCPMM provides byte-addressable access granularity which differentiates it from similar technologies which were limited to larger block-based accesses. This creates new opportunities for low latency SCM usage in data centers as either a volatile memory extension to DRAM or as a low latency persistent storage media.

2.2.1 Operation mode overview

DCPMM can be configured to operate in one of two different modes: Memory Mode and App Direct Mode [38]. Illustrations of the modes are shown in Figure 2.

In **Memory Mode**, as shown in Figure 2, the DRAM capacity is hidden from applications and serves as a cache for the most frequently accessed addresses, while DCPMM capacity is exposed as a single large volatile memory region. Management of the DRAM cache and access to the DCPMM is handled exclusively by the CPU’s memory controller. In this mode applications have no control of where their memory allocations are physically placed (DRAM cache or DCPMM).

In **App Direct Mode**, DRAM and DCPMM will be configured as two distinct memories in the system and are exposed separately to the application and operating system. In this case, the application and OS have full control of read and write accesses to each media. In this mode, DCPMM can be configured as block-based storage with legacy file systems or can be directly accessed (via DAX) by applications using memory-mapped files.

2.3 Facebook RocksDB workloads

For our experiments and evaluation we chose the largest RocksDB use cases at Facebook, which demonstrate typical uses of key-value storage ranging from messaging services to

large storage for processing realtime data and metadata. Note than these are not the only workloads that benefit from our designs. The descriptions of the services are as follows:

WhatsApp: With over a billion active users, WhatsApp is one of the most popular messaging applications in the world [39]. WhatsApp utilizes ZippyDB as its remote data store. ZippyDB [40] is a distributed KV-store that implements Paxos on top of RocksDB to achieve data reliability and persistence.

Tectonic Metadata: The Tectonic Metadata databases are also stored in ZippyDB and are an integral part of the large blob storage service of Facebook that serves billions of photos, videos, documents, traces, heap dumps, and source code [41, 42]. Tectonic Metadata maintains the mappings between file names, data blocks and parity blocks, and the storage nodes that hold the actual blocks. These databases are distributed and fault-tolerant.

Laser: Laser is a high query throughput, low (millisecond) latency, peta-byte scale key-value storage service built on top of RocksDB [43]. Laser reads from any category of Facebook’s real-time data aggregation services [44] in real-time or from a Hadoop Distributed File System [45] table daily.

3 Hybrid cache design choices

3.1 The challenges of SCM deployment

The first challenge of introducing SCM in RocksDB is identifying which of its components to map to SCM. We chose the uncompressed block cache because it has the largest memory usage in our RocksDB workloads and because our studies reveal that a number of our production workloads, which are read-dominated, benefit from optimizing read operations by block cache extension. We also focused on the uncompressed block cache instead of the compressed ones, so that we can minimize CPU utilization increase when performing compression/decompression. This allowed us to increase the size of SCM (block cache) without requiring additional CPU resources. We also chose block cache over memtable because SCM provides better read bandwidth than writes, hence helping our read-demanding workloads. We then expanded the block cache size by utilizing SCM as volatile memory. We chose this approach because extending the memory capacity while reducing the size of DRAM and the cost of our servers is the primary goal. Although we can benefit from persisting block cache and memtable in SCM for fast cache warmup and fast write access, we left this for future work.

The next challenge is, how we should configure SCM to get the best performance. We have the options of using memory mode, that does not require software architecture changes or app-direct mode that necessitates modification in RocksDB but provides control of DRAM and SCM usage. Figure 3 demonstrates how memory-mode compares to our optimized app-direct mode. Optimized app-direct mode with various DRAM and SCM sizes, renders 20-60% throughput improvement and 14-49% lower latency compared with memory mode. This insight supports that our optimized implementation has

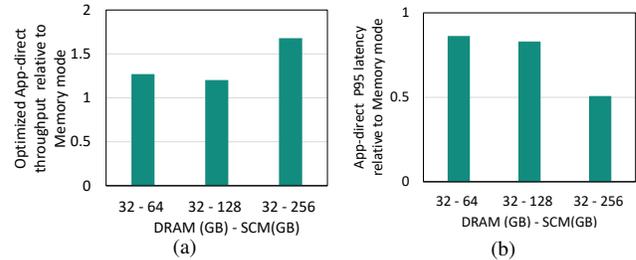


Figure 3: Throughput and latency comparison for memory mode and our optimized hybrid-cache in app-direct mode for WhatsApp.

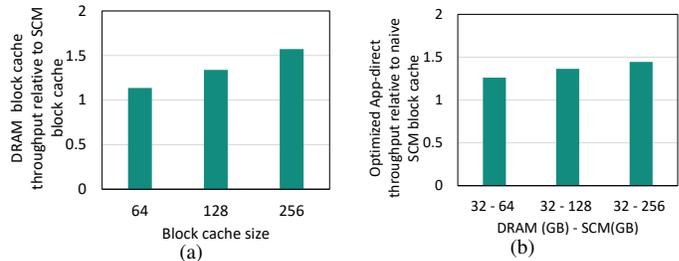


Figure 4: (a) Throughput of DRAM vs SCM based block cache. (b) Throughput of naive SCM vs optimized hybrid-cache for WhatsApp.

a better caching mechanism than memory mode, hence we focused our analysis on app-direct mode.

With app-direct we can manage the allocation of RocksDB’s components (memtable, data, filter, and index blocks) to DRAM or SCM. But since we know the data access latency of SCM is slower than DRAM (see Table 1), we have to consider its effect. We compared the throughput of allocating the block cache to DRAM or SCM in app-direct mode in vanilla RocksDB to understand the impact of the higher SCM access latency. As seen in Figure 4a, the slower SCM latency creates 13%-57% difference in throughput when we compare DRAM based block cache to a naive SCM block cache using app-direct mode. This result guided us to carefully utilize DRAM and SCM in our designs. In single-socket machines such as *IP servers*, we have one CPU and 32GB - 64GB DRAM capacity. Out of this DRAM, memtable, index, and filter blocks consume 10-15 GBs. The rest of DRAM and the additional SCM can be allocated for block cache. We compared the naive SCM block cache implementation (all block cache allocated to SCM using app-direct) to a smarter and optimized hybrid cache, where highly accessed data is allocated in DRAM and the least frequently access in SCM. The results in Figure 4b show with optimized app-direct we achieve up to 45% better throughput compared to a naive SCM block cache. From this, we can determine that implementing a hybrid cache compensates for the performance loss due to the higher SCM access latency. These results together with the high temporal locality of our workloads (as discussed below) motivated us to investigate a hybrid cache.

3.2 RocksDB workload characteristics

Below we scrutinize the characteristics of our largest RocksDB workloads that guided our hybrid cache design.

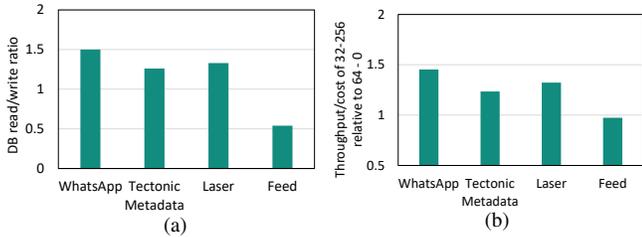


Figure 5: (a) Read to write ratio to DB. (b) Key-value throughput/-cost compare for large block cache friendly workloads and Feed with higher writes than reads.

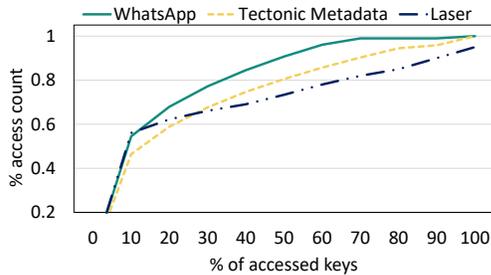


Figure 6: Key-value locality.

Reads and writes to DB: As we discussed earlier, prior work showed that optimizing reads provides large impact in commercial data center workloads [25–27]. Our studies also show that we have a large number of read-dominated workloads, therefore optimizing the block cache, used for storing data for fast read access, will benefit a number of our workloads. In RocksDB, when a key is updated in memtable it will be invalid in the block cache. Hence, if the workload has more write queries than reads, then the data in the cache will become stale. Note that write-dominated workloads won’t be affected by our hybrid cache designs because we did not reduce any DRAM buffer (memtable) in the write path. In our studies, we profiled deployed RocksDB workloads for 24 hours using an internal metrics collection tool to comprehend the read and write characteristics. Figure 5a shows the workloads described in Section 2.3 reads more bytes from the DB than it writes. To contrast, we evaluated one of our write-dominated workloads, Feed, also seen in Figure 5a. In Figure 5b, we calculated the throughput per cost of *IP server variants* with 32 GB DRAM and 256 GB SCM capacity normalized to throughput/cost of *IP server* with 64 GB DRAM capacity. The throughput/cost improvement of Feed for our largest DRAM-SCM system cannot offset the additional cost due of SCM. Hence, we focus on exporting read-dominated workloads to our hybrid systems.

Key-value temporal locality: Locality determines the cacheability of block data given a limited cache size. A hybrid cache with a small DRAM size will only benefit us if we have a high temporal locality in the workloads. In this case, significant access to the block cache will come from the DRAM cache, and SCM will hold the bulk of less frequently accessed blocks. We used RocksDB trace analyzer [35] to investigate up to 24 hours query statistics of workloads running on production *2P server* and evaluate locality as the distribution

of the total database access counts to the total keys accessed per database. Figure 6 shows that our workloads possess a power-law relationship [46] between the number of key-value pair access counts and the number of keys accessed. We can observe in the figure that 10% of the key-value pairs carry ~50% of the key-value accesses. This makes a hybrid cache design with small DRAM practical for deployment.

DB and cache sizes: The desirable DRAM and SCM cache sizes required to capture workload’s locality is proportional to the size of the DB. Workloads with high key-value locality and large DB sizes can achieve a high cache hit rate with limited cache sizes. But as the locality decreases for large DB sizes, the required cache sizes will grow. In the extreme case of random key-value accesses, all blocks will have similar heat, diluting the value of the DRAM cache and reducing overall hybrid cache performance asymptotically toward that of the SCM-only block cache. For small DBs, locality might not play a significant role because the majority of the DB accesses fit in a small cache. Such types of workloads will not be severely affected by DRAM size reduction, and choosing the *IP server variants* with large SCM capacity will be a waste of resources. In our studies, after looking at various workloads in production, we choose our hybrid DRAM-SCM cache configuration to accommodate several workloads with larger DB sizes (~2TB total KVstorage per server).

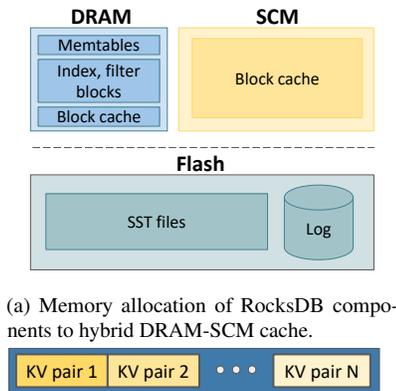
4 DRAM-SCM hybrid cache module

In our RocksDB deployment, we placed the memtables, index blocks, and filter blocks in DRAM. We then designed a new hybrid cache module that allocates the block cache in DRAM and SCM. The database SST files and logs are located in Flash. The overview of RocksDB components allocation in the memory system is shown in Figure 7a. Our goal in designing the new hybrid cache module is to utilize DRAM and SCM hierarchically based on their read access latency and bandwidth characteristics. In our design, we aim to place hot blocks in DRAM for the lowest latency data access, and colder blocks in SCM as a second tier. The dense SCM based hybrid block cache provides a larger effective capacity than practical with DRAM alone leading to higher cache hit rates. This dramatically decreases IO bandwidth requirements to the SST files on slower underlying flash media.

The block cache is an integral data structure that is completely managed by RocksDB. Similarly, in our implementations, the new hybrid cache module is fully managed by RocksDB. This module then is an interface between RocksDB and the DRAM and SCM block caches, and fully manages the caches’ operations. The overall architecture of the hybrid cache is shown in Figure 7c. The details of its internal components are as follows:

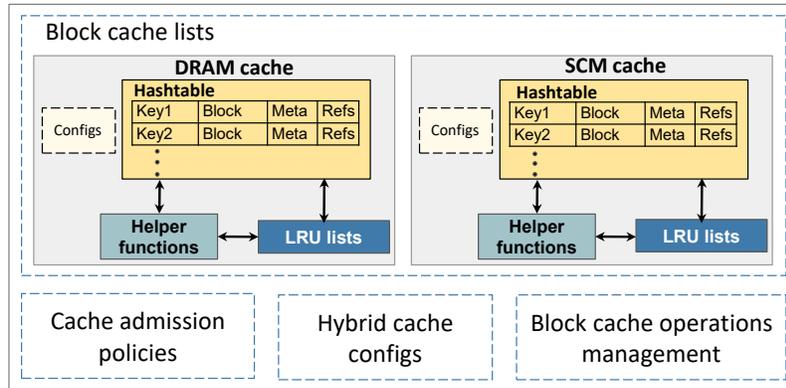
4.1 Block cache lists

The hybrid cache is a new top-level module in RocksDB that maintains a list of underlying block caches in different tiers.



(a) Memory allocation of RocksDB components to hybrid DRAM-SCM cache.

(b) Data block structure.



(c) Hybrid tier cache component and architecture.

Figure 7: RocksDB components and memory allocation.

The list of caches are extended from the existing RocksDB block cache with LRU replacement policy. Note that in our implementations we have DRAM and SCM cache, but the module can manage more than these two caches such as multiple DRAM and SCM caches in a complex hierarchy.

4.1.1 Block cache architecture and components

The internal structures of DRAM and SCM caches, which are both derived from the block cache, are shown in Figure 7c. The block cache storage is divided into cache entries and tracked in a **hashtable**. Each cache entry holds a key, data block, metadata such as key size, hash, current cache usage, and a reference count of the cache entry outside of the block cache. The data block is composed of multiple key-value pairs as shown in Figure 7b. Binary searches are performed to find a key-value pair in a data block. The data block size is configurable in RocksDB. In our case, the optimal size was 16KB. As the number of index blocks decreases we can increase the data block size. As a result, with 16KB we were able to reduce the number of index blocks making room for data blocks within our limited DRAM capacity. Every block cache has **configs** that are configured externally. This includes size, a threshold for moving data, a pointer to all other caches for data movement, and the memory allocator for the cache. The cache maintains an **LRU list** that tracks cache entries in order from most to least recently used. The **helper functions** are used for incrementing references, checking against the reference threshold, transferring blocks from one cache to another, checking size limits, and so on. For the components listed above we extended and modified RocksDB to support tiered structure, different kinds of admission policies and we designed new methodologies to enable data movement between different caches and to support memory allocation to different memory types.

4.1.2 Data access in the block cache

A block is accessed by a number of external components to the block cache, such as multiple reader clients of the RocksDB database. The number of external referencers is tracked by the reference count. Mapping to a block is created when it is

referenced externally, this will increment the reference count. Whereas when the referencer no longer needs a block, mapping is released, and the reference count is decremented. If a block has zero external references, it will be in the hashtable and tracked by the LRU list. If a block gets referenced again, then it will be removed from the LRU list. Note that in the LRU list, newly released blocks with no external references are on the top of the LRU list as the most recently used blocks, and when blocks are evicted, the bottom least recently used blocks are evicted first. The block cache is used for read-only data, hence it doesn't deal with any dirty data management. Therefore, when transferring data between DRAM and SCM we do not have to deal with dirty data.

4.2 Cache admission policies

Identifying and retaining blocks in DRAM/SCM based on their access frequencies requires proactive management of data transfer between DRAM, SCM, and flash. Hence, we developed the following block cache admission policies.

4.2.1 DRAM first admission policy

In this admission policy, new blocks read from flash are first inserted into the hashtable of the DRAM cache. The block cache data structures are size limited. Hence when the size of the blocks allocated in the DRAM cache exceeds the size limits, the oldest entries tracked by the DRAM LRU list are moved to the next tier cache (SCM cache) by the data mover function of the DRAM cache, using the SCM cache's memory allocator. On lookups, both the DRAM and SCM caches are searched until the cache block is found. If it is not found, it will initiate a flash read. Similar to the DRAM cache when the capacity of the SCM cache exceeds the limit, the oldest entries in the LRU list of the SCM cache are freed to accommodate new cache blocks evicted from the DRAM cache.

4.2.2 SCM first admission policy

In this admission policy, new blocks read from flash are first inserted in the hashtable of the SCM cache. Unlike the DRAM first admission policy, this policy has a configurable threshold for moving data from the SCM cache to the DRAM cache.

When the external references of cache entries in the SCM cache surpasses the reference threshold, blocks are considered to be hot and will be migrated to the DRAM cache for faster access. The data movement, in this case, is handled by the data mover function of the SCM cache. When the capacity of both the DRAM and SCM caches are full, the oldest LRU blocks are evicted from both caches. In the DRAM cache, LRU entries are moved back to the SCM cache, whereas in the SCM cache, the LRU entries are freed to accommodate new block insertions. On lookup, both the DRAM and SCM caches are searched until the cache block is found.

4.2.3 Bidirectional admission policy

In Bidirectional admission policy, similar to the DRAM first admission policy, new data blocks are inserted into the DRAM cache. As the capacity of DRAM and SCM cache reach the limit, the oldest LRU entries are evicted to SCM cache from the DRAM cache and are freed for the case of SCM cache. The difference between DRAM-first and Bidirectional cache is, after the oldest LRU entries are evicted from DRAM to SCM cache, if the external reference to an entry surpasses a preset threshold it is transferred back to the DRAM cache. This property allows us to re-capture fast access performance for blocks with inconsistent temporal access patterns.

In the hybrid cache, we can set the three of the admission policies or we can easily extend a new policy by configuring how to insert, lookup, and move data in the list of block caches. These configs are global parameters in the top-level hybrid cache and are used by the block cache operations manager and list of block caches. Optionally the thresholds for moving data in SCM first and Bidirectional policies can be set to change values based on the current usage of the caches. But in our experiments, we didn't see benefit with changing values. We also performed an analysis with different sizes of cache thresholds and we show the optimal threshold for SCM first and Bidirectional in our evaluations.

4.3 Hybrid cache configs

The hybrid cache configurations are set outside of the module by RocksDB, and include pointers to configs of all block caches, the number of block caches, ids, tier numbers of the caches, and admission policy to use. Configs are used during instantiation and at run time to manage database operations.

4.4 Block cache operation management

This unit redirects external RocksDB operations such as insert, lookup, update, and so on to the target block cache based on the admission policy. For example, it decides if an incoming insert request should go to the DRAM or SCM cache.

5 Systems setup and implementation

5.1 DRAM-SCM cache implementation

We configured Intel DCPMMs in App Direct mode using the IPMCTL [47] tool in our experiments. We used Linux Kernel

5.2 that brings support for a volatile use of DCPMM by configuring a hot-pluggable memory region called KMEM DAX. We then used NDCTL 6.7 [48], a utility for managing SCM in the Linux Kernel, to create namespaces in DCPMM in devdax mode. This mode provides direct access to DCPMM and is faster than filesystem-based access. We then used DAXCTL [48] utility for configuring DCPMM in system-ram mode so that DCPMM will be available in its own volatile memory NUMA node. To implement a hybrid DRAM-SCM cache we used memkind library [49], which enables partitioning of the heap between multiple kinds of memories such as DRAM and SCM in the application space. After the system is configured with DRAM and DCPMM memory types, we modified RocksDB block cache to take two types of memory allocators using memkind. We use MEM_KIND_DAX_KMEM kind for SCM cache and a regular memory allocator for DRAM. The overview of our implementation is shown in Figure 8.

Table 2: System setup: hardware specs and software configs.

Specification	System config
Application version	RocksDB 6.10
OS	CentOS-8
Linux kernel version	5.2.9
CPU model	Intel(R) Xeon(R) Gold 6252 @ 2.10GHz
Socket/Cores per socket	2/24
Threads total	96
L1/L2 cache	32 KB
L2/L3 cache	1 MB/35.75 MB
Memory controllers total	4
Channels per controller	3
Slots per channel	2
DRAM size	DDR4 192 GB (16 GB X 12 DIMM slots)
SCM size	DDR-T 1.5 TB (128 GB X 12 DIMM slots)
SSD model	Samsung 983 DCT M.2 NVMe SSD
SSD size/filesystem	2 TB / xfs

5.2 Evaluation hardware description

In our evaluations we used a Intel Wolf Pass [50] based system, utilizing 2 CPU sockets populated with Intel Cascade Lake processors [51]. Each CPU has 2 memory controllers with 3 channels each, and 2 DIMM slots per channel, for a total of 24 DIMM slots. DIMM slots were populated by default in a 2-2-2 configuration, with 12 total *16GB PC4-23400 ECC Registered DDR4 DRAM DIMMs* and 12 total *128GB Intel® Optane™ PMem 100 DIMMs*. This makes up a total of 192GB DRAM and 1.5TB of SCM per system. Backing storage for the RocksDB database was a *Samsung 983 DCT M.2 SSD*. The detailed specification is listed in Table 2.

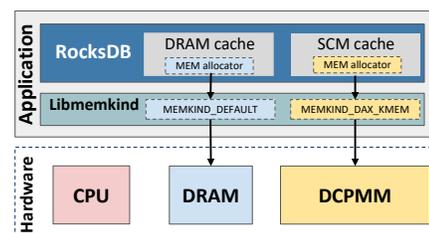


Figure 8: DRAM-SCM cache configuration with libmemkind.

Table 3: OCP Tioga Pass (2P server) and OCP Twin Lakes Platforms (1P server) details.

Specification	OCP Tioga Pass Config	OCP Twin Lakes Config
CPU model	Xeon(R) Gold 6138	Xeon(R) D-2191
Sockets/cores per socket	2/20	1/18
Threads total	80	36
Memory controllers total	4	2
Channels per controller	3	4
DRAM size	DDR4 256 GB	DDR4 64 GB
SSD size	2 TB	2 TB

5.3 Facebook server designs

The Facebook platforms used for TCO analysis are the *2P server* platform based on the OCP Tioga Pass specification [52], and the *1P server* platform based on the OCP Twin Lakes specification [53]. In addition, we propose several *1P server* variants utilizing differing capacities of DRAM and SCM. The detailed specifications and relative costs of these platforms compared to the baseline *2P server* platform are listed in Table 3 and 4. Platform costs are calculated based on current OCP solution provider data [54,55] and DCPMM cost relative to DRAM provided in [56,57]. The relative cost of DRAM and DCPMM are predicted to maintain similar trends over time [56]. We calculated the cost by adding the cost of individual modules. For DRAM and SCM, we used 16GB and 128GB modules, respectively. In the TCO calculations although introducing SCM adds additional power cost, when we consider the overall power of the system including CPU, NIC, and SSD, the increase of power even for our largest 1P server becomes minimal. We have a power budget for a rack of servers with some power slack and the slight rise in power for SCM is sufficiently below our rack power budget.

Table 4: DRAM and SCM cache in server configuration and memory sizes used block cache in our experiments.

Configuration	2P serv.	1P serv.	64-128	32-128	32-256
DRAM size	256	64	64	32	32
SCM size	0	0	128	128	256
DRAM Block cache size	150	40	40	12	12
SCM Block cache size	0	0	100	100	200
Other Rocksdb components	10-15	10-15	10-15	10-15	10-15
Codebase	5	5	5	5	5
2P rel. cost	1	0.43	0.5	0.46	0.53

5.4 Cache and memory configurations

To experiment with SCM benefits when added to existing configurations we examined server configurations with different sizes of DRAM and SCM. The configurations we used are shown in Table 4. Our experiments verified that *1P server* has significant performance loss compared to *2P server*. The prominent questions here are how much benefit can we achieve by adding SCM to *1P servers*, and can we still maintain the TCO benefits of *1P server* platform. Hence

we used the *1P server* with 64 DRAM and no SCM as the baseline for our evaluations. We then evaluated how much performance we can gain by adding 128 GB SCM to the baseline. Since we are interested in DRAM constrained server configurations, we also evaluate the performance gain when we further reduce DRAM to 32 GB while adding 128 or 256 GB SCM. This gives us the four server configurations provided in Table 4. Although we experimented with 64GB of SCM, as seen in Section 3, to understand the performance of different SCM sizes, DCPMM is not available as a 64GB module, so we didn't consider it in the evaluation below. To run all of the server configurations, we limited the memory usage of the DRAM and SCM for the workloads to the sizes given in Table 4. We also use 1 CPU in our experiments because *1P servers* are single-socket machines. We aimed to maximize block cache allocation to evaluate our DRAM-SCM hybrid cache. To do this we studied the memory usage of other components in RocksDB when it runs in our production servers. We then subtracted these usages from total memory and assigned the rest of the memory for the Block cache. The block cache sizes we used in our evaluations are illustrated in Table 4.

5.5 Workload generation

To generate workloads, we first selected random sample hosts running WhatsApp, Tectonic Metadata, and Laser in production deployments to collect query statistics traces, being careful to select leaders for use cases relying on Paxos. Note that hosts running the same services manifest similar statistics. Then we followed the methodology in [28,35] to model the workloads. We also extended these methodologies [28,35] to scale workloads to match the production deployments. The *db_bench* workloads we developed mirror the following characteristics of production RocksDB workloads.

KV-pair locality: This is characterized by fitting real workload trace profiles to a probability cumulative function using power distributions in MATLAB [58] based on the power-law characteristics of the workloads (see Figure 6).

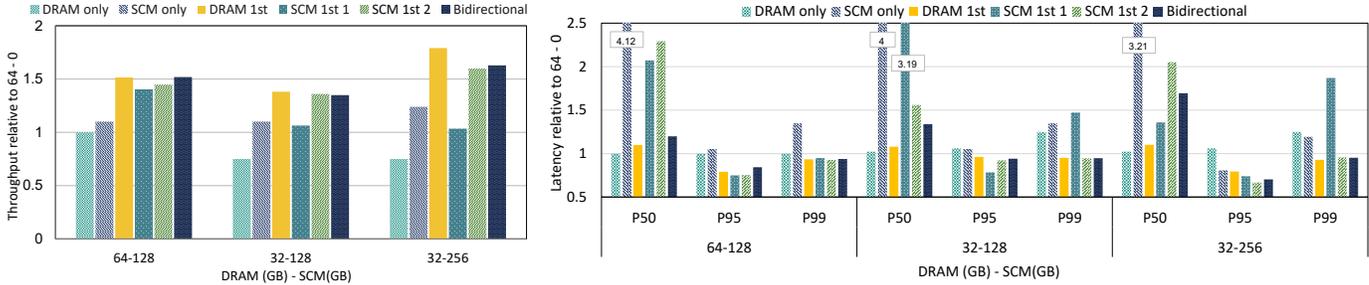
Value distribution: The granularity of value accessed is modeled using Pareto distribution from workload statistics in MATLAB [59].

Query composition: The percentage of get, put, and seek queries are incorporated in the *db_bench* profiles.

DB, key, and value sizes: We added the average values of the number of keys per database, key, and value size from collecting data from our production servers to create a realistic DB sizes in the *db_bench* profile.

QPS: The QPS in *db_bench* is modeled using sine distributions [60] based on trace collected in the production host.

Scaled db_bench profiles: After generating the workloads *db_bench* profile for a single database we scaled the workloads by running multiple RocksDB instances, simulating the number of production shards per workload on a single host. These shards share the same block cache. In RocksDB there exists multiple readers and writers to the database. To imitate



(a) Throughput comparison of admission policies. (b) Latency comparison of admission policies.

Figure 9: Throughput and application read latency comparison using only DRAM for block cache, using only SCM for block cache, and hybrid DRAM-SCM admission policies (DRAM first, SCM first, and Bidirectional) for WhatsApp using all server configurations in Table 4. (a) Throughput comparisons (db operations/sec) for admission policies. Here the baseline is 64-0 configs. (b) P50, P95, and P99 latencies for all admission policies and server configs compared to 64-0 server.

this property we run multiple threads reading and writing to the set of shards in the db_bench process.

6 Evaluation

6.1 Throughput and latency comparison for admission policies

In Figure 9 we compare the throughput achieved for 5 different categories: **DRAM Only**: The Block Cache is only allocated in DRAM. **SCM Only**: The Block Cache is only allocated in SCM using app-direct mode. **DRAM First**: The DRAM first policy discussed in Section 4. **SCM First 1 and 2**: The SCM first policy discussed in Section 4 with two threshold values. SCM 1st 1, with threshold value of 2 and SCM 1st 2 with the optimal threshold which is 6. **Bidirectional**: The Bidirectional policy discussed in Section 4 with the optimal threshold value which is 4.

In Figure 9a, we demonstrate the throughput differences of all admission policies for WhatsApp. The results show that using only SCM for block cache provides 15% throughput improvement for the 128 GB SCM configurations and 25% improvement for the 256GB SCM compared to the baseline (a server configuration with 64GB DRAM and no SCM). If we look at Figure 9b, because of latency differences between SCM and DRAM, getting data only from SCM worsens the P50 application read latencies. Therefore we conclude that while the default RocksDB SCM implementation may decrease flash IO utilization, it will have a net negative impact on Facebook application performance due to the 2X - 4X worse P50 latency observed. We can also see in the figure for all the server configurations DRAM first policy achieves the best performance. For 64 - 128 and 32 - 128 configurations, SCM first and Bidirectional get close in throughput benefit to DRAM first, but when there is a large block cache, like in the 32 - 256 configuration DRAM first attains the best result. This is because, in DRAM-first, data transfer between DRAM and SCM cache only occurs once, when DRAM cache evicts data to SCM. But in the case of SCM and Bidirectional policies, data transfer occurs when DRAM evicts data to SCM and when hot blocks are transferred from SCM to DRAM.

This creates more bandwidth consumption across the DDR bus resulting in performance degradation, especially for configurations with large block cache sizes. For larger DRAM capacities, SCM first 1, SCM first 2, and Bidirectional policies have comparable throughput because the large DRAM size reduces data evictions to SCM. But as DRAM is reduced (in 32 - 128), SCM 1st throughput falls quickly because it will move data from SCM to DRAM with a low activation threshold. When we increase DCPMM capacity in the 32 - 256 case, data transfer increases even for SCM 2 and Bidirectional policies, hence the DRAM first policy is the overall performance winner. If we look at read latencies shown in Figure 9b, the P50 latency remains similar for DRAM first compared to 64 - 0 configurations. The reason for this is that P50 latencies are primarily governed by DRAM accesses. The effect of data transfer from flash instead of SCM can be observed in the P95 and P99 latencies, where the DRAM First policy does significantly better than other policies and the default configuration with no SCM. Tectonic Metadata and Laser (not shown here) also attain the best performance with DRAM first policy.

6.2 Performance comparison of DRAM first policy for all workloads

Figure 10 shows the throughput and latency comparison of WhatsApp, Tectonic Metadata, and Laser for DRAM first admission policy (**the best admission policy for all workloads**) for all server configurations. As seen in the figure, our hybrid block cache implementation provides throughput improvement for all the workloads. As seen in Figure 10a, 10b, and 10c throughput is increased up to 50 - 80% compared to the baseline *IP server's* 64 - 0 due to the addition of SCM. The throughput increase is correlated to the total size of the block cache. Note that increasing the SCM or DRAM capacity further than 256GB will require either more DIMM slots or higher density DIMMs, with different price/performance/reliability considerations. The size of the database also impacts locality and the maximum throughput benefit, as discussed in 3.2. Because Tectonic Metadata has a larger DB size than WhatsApp or Laser the throughput benefit is expected to be smaller for each configuration. Looking at application level

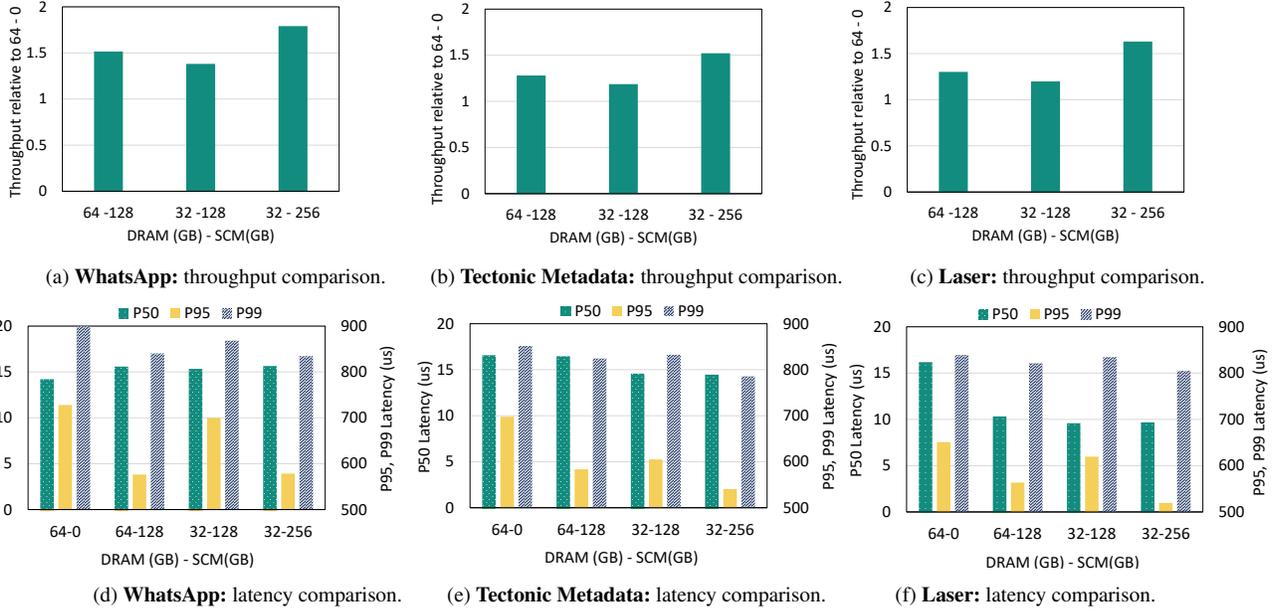


Figure 10: Throughput and application read latency comparison for WhatsApp, Tectonic Metadata and Laser for DRAM first admission policy using all server configurations shown in Table 4. (a,b, and c) Throughput comparisons (db operations/sec). Here the baseline is 64 - 0 configs. (d, e and f) absolute P50, P95, and P99 latencies for all workloads and server configs.

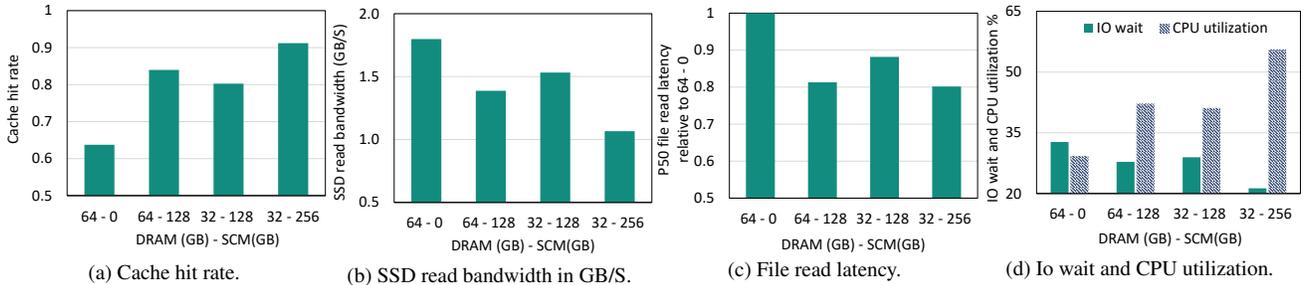


Figure 11: Cache utilization, IO read bandwidth, IO read latency, IO wait and CPU utilization of WhatsApp for DRAM 1st admission policy.

read latency in Figure 10d, 10e, and 10f, we observe that P50 latency is relatively stable for WhatsApp and Tectonic Metadata. While P50 latency does improve for Laser, the absolute magnitude of the improvement is less significant than the improvements to tail latency. P95 and P99 show an overall improvement of 20% and 10% respectively for all services. The P50 latencies primarily reflect situations where the data is obtained from DRAM. The benefit of SCM is reflected in P95 and P99 scenarios where in one case the data is in SCM, while the default case the data is in flash storage. Write latencies in all cases stay similar since we are only optimizing the block cache used for reads, while writes always go to the memtables residing in DRAM.

6.3 IO bandwidth, cache and CPU utilization

Figure 11 illustrates the cache hit rate, IO bandwidth, and latency improvement of DRAM first policy for WhatsApp. As seen in Figure 11a, the higher capacity of the block cache (sum of DRAM and SCM cache) leads to a higher cache hit rate. We show in Figure 11a, that for WhatsApp the hit

rate increases up to 30% and the increase is correlated to the cache size. Tectonic Metadata and Laser (not shown here) also follow a similar pattern of increasing hit rates.

Another important indicator explaining throughput gain is SSD bandwidth utilization. As the cache hit rate increases for the server configurations with SCM it translates to less demand for read access from the SSD, and therefore decreased IO read bandwidth. Figure 11b shows for WhatsApp adding SCM reduces SSD read bandwidth by up to 0.8 GB/s, or roughly 25% of the SSD's datasheet max read bandwidth. Figure 11c shows the file read latency improvement for all server configurations relative to 64 - 0. Decreased demand for read IO bandwidth improves the P50 latency by up to 20%. Latencies at higher percentiles stay the same because there are still scenarios where the IO queue will be saturated with reads, which drives worst-case latency. But for the majority of the requests, latencies are improved because of the decrease in IO bandwidth. The other workloads also show similar patterns. Figure 11d shows the CPU utilization for all server configurations. We observe that the CPU utilization increases

as the block cache size increases. This is due to the increase in CPU activity as we increase amount of data accessed with low IO wait latency from the cache. One thing to note is, even though CPU utilization increase for our new *IP server variants*, we can still safely service the workloads with 1 CPU even in the largest 256 SCM configuration.

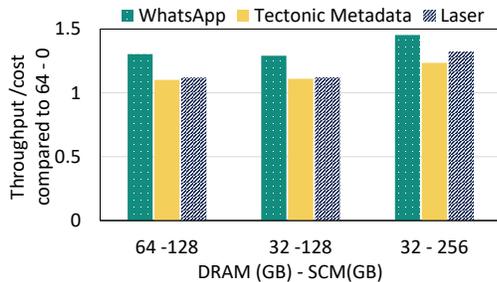


Figure 12: Throughput/cost of WhatsApp, Tectonic Metadata and Laser normalized to 64 - 0 IP servers throughput/cost.

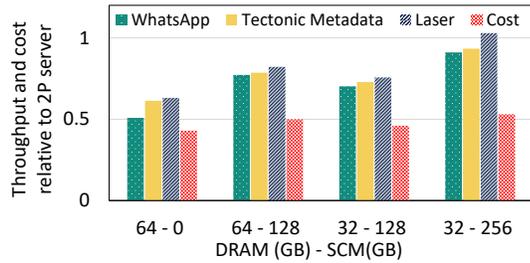


Figure 13: Throughput of IP and its variants compared to 2P servers.

6.4 Cost and performance

In the above sections, we aim to understand the performance achieved for different DRAM and SCM variations of the *IP server* configuration. The large capacity of SCM per DIMM slot enables us to dramatically increase the memory capacity of the platform without impacting the server motherboard design. As seen in Table 4 adding SCM increases the cost of a server. Figure 12 estimates the performance per cost compared to baseline *IP server* (64 - 0) configuration. We observe in the figure that the 32 - 256 configuration gives the best cost relative to performance across all workloads. In this configuration the 23% cost increase over the 64 - 0 baseline produces a 50% - 80% performance improvement. To a smaller degree the 64-128 and 32-128 configurations also provide performance-relative cost benefits over the standard *IP server*. Notable is the fact that the benefit across these two configurations is nearly identical due to the proportional difference in DRAM cost vs. performance increase. If future DRAM/SCM hardware designs provide additional flexibility across capacity pricing then we may discover new configurations which achieve even larger TCO benefits.

In Figure 13, we present a throughput comparison between the *2P server* and the various *IP server* configurations. The figure shows that increasing the amount of SCM brings throughput closer to parity with the *2P server* for a minimal

Table 5: Performance equivalent TCO relative to 2P.

Server types	2P server	1P server	1P (32 - 256)
Relative TCO	1.0	0.72 - 0.86	0.52 - 0.57

increase in relative cost. While the baseline *IP server* (64 - 0) configuration only achieves 50 - 60% of the performance of a *2P server*, the 32 - 256 *IP* variant raises relative performance to 93 - 102%. By dividing the relative cost of the platforms (Table 4) by their relative performance (Figure 13) for each workload we derive the performance-equivalent TCOs in Table 5. In the case of the 32 - 256 configuration, improving *IP* performance with SCM improves the relative TCO to 0.52 - 0.57 of the *2P server*. Therefore, we demonstrate that deploying SCM configurations of *IP servers* instead of *2P servers* results in an **overall cost savings of 43% - 48%** across some of the largest RocksDB workloads at Facebook. Note that although SCM adds more power per rack (5-10%) the performance gain allows us to deploy fewer racks, hence it's evidently a power win compared to the 64-0 server.

General takeaway: From this project, we have learned that SCM creates cost-effective solutions to increase performance in data centers. We have observed that based on the workload hotness/locality we can utilize DRAM and SCM to serve most workloads efficiently.

Even though the performance and cost of using SCMs are impressive for the chosen workloads, and the performance gain can be translated to other read-dominated workloads in our environment, a discussion on whether we should have a deployment of SCMs in Facebook at scale is outside the scope of this paper. We briefly discuss some additional challenges of mass-scale deployment:

Workloads: Section 3.2 talks about the class of applications that would benefit from SCM. We have also identified a number of write-heavy workloads at Facebook that would not benefit from SCMs.

Reliability: Since SCMs are not widely available in the market, the reliability of SCMs is a concern until they have been proven in mass deployments.

SKU diversification: Adding a new hardware configuration into the fleet requires consideration of other costs like maintaining a separate code path and creating a new validation and sustainability workflow. This complexity and cost will be added to the practical TCO of any new platform deployment.

7 Discussion and future work

In the previous section, we demonstrated that KV Stores based on RocksDB are examples of the potential advantages of SCM in a large data center. In the future, we want to experiment with extending the memory capacity of other large memory footprint workloads using SCM. Some candidate large scale workloads that will profit from large memory capacity are Memcached [61, 62] and graph cache [27]. Memcached is a distributed in-memory key-value store deployed in large data centers. Optimizing Memcached to utilize SCM will enable an extension of memory beyond the capacity of DRAM.

Graph cache is a distributed read optimized storage for social graphs that exploits memory for graph structure-aware caching. These workloads are read-dominated and have random memory accesses that can benefit from the high density and byte-addressable features of SCM. Although in this paper we did not leverage the persistent capability of SCM for RocksDB uncompressed block cache, in the future we want to study the benefits of fast persistent SCM for the memtable and SST files. We also want to explore with SCM is its performance via emerging connectivity technology such as Compute Express Link (CXL) [63]. The workloads we analyzed in this paper are more latency bound than memory bandwidth bound but, for high memory bandwidth-demanding services, sharing DRAM and SCM on the same bus will create interference. In such cases having dedicated SCM access via CXL will avoid contention, but at the same time will potentially increase data access latency, requiring careful design consideration.

8 Related work

Performance analysis and characterization in DCPMM:

Recent studies proved the potential of commercially available Intel® Optane™ memory for various application domains. [19, 64] has determined the performance improvement of DCPMM in memory and app-Direct modes for graph applications. [20, 21] evaluated the performance of DCPMM platform as volatile memory compared to DRAM for HPC applications. DCPMM performance for database systems were shown in [16, 65–68] both as a memory extension and for persisting data. Works such as [15, 18, 24, 69] also shows the characteristics and evaluations of DCPMM when working alone or alongside of DRAM. Specifically, [18] has identified the deviation of DCPMM characteristics from earlier simulation assumptions. While these works shed light on the usage of DCPMM for data-intensive applications, in our work, based on the memory characteristic findings of these work, we analyzed the performance of DCPMM for large data data-center production workloads. Our work focus on utilizing the DCPMM platform to the best of its capability and study its possible usage as a cost-effective memory extension for future data-center system designs.

Hybrid DRAM-SCM systems: Previous works studied hybrid DRAM-SCM systems to understand how we can utilize these memories with different characteristics, together and how they influence the existing system and software designs. [70–73] have shown the need for a redesign of existing key-value stores and database systems to take into account the access latency differences between DRAM and SCM. Similarly, by noting the latency differences in these memories, we carefully place hot blocks in DRAM and colder blocks in SCM in our implementations. When deploying hybrid memory, another question that arises is, how to manage data placement between DRAM and SCM. In these aspects, [74] demonstrated efficient page migration between DRAM and SCM based on the memory access pattern observed in the

memory controller. In addition, [75–83] perform data/page transfer by profiling and tracking information such as memory access patterns, read/write intensity to a page/data, resource utilization by workloads, and memory features of DRAM and SCM, in hardware, OS, and application level. These works aim to generalize usage of DRAM and SCM to various workloads without involving the application developer, hence requiring hardware and software monitoring that is transparent to the application developers. But in our case, the RocksDB application-level structure exposes separate reads and writes paths and frequency of access of data block. These motivated us to implement our designs in software without requiring any additional overhead in the OS and hardware.

RocksDB performance improvements: [84] demonstrated how to decrease the memory footprint of MyRocks, which is built on top of RocksDB, using block access based nonvolatile memory (NVM) by implementing secondary block cache. While their methods also decrease DRAM size required in the system, the block-based nature of NVM increases read amplification. This is because, the key-value size in RocksDB is significantly less than the size of the block, whereas in our methods, using byte addressable SCM avoids such issues.

9 Conclusion

The increasing cost of DRAM has influenced data centers to design servers with lower DRAM per compute ratio. These servers have shown to decrease the TCO for scalable workloads. Nevertheless, this type of system design diminishes the performance of large memory footprint workloads that relies on DRAM to cache hot data. Key Value stores based on RocksDB is one such class of workloads that is affected by the reduction of DRAM size. In this paper, we propose using Intel® Optane™ PMem 100 Series SCMs (DCPMM) in AppDirect mode to extend the available memory for RocksDB to mitigate performance loss in smaller DRAM designs while still maintaining the desired lower TCO of smaller DRAM systems. We carefully studied and redesigned the block cache to utilize DRAM for the placement of hot blocks and SCM for colder blocks. Our evaluations show up to 80% improvement to throughput and 20% improvement in P95 latency over the existing small DRAM platform when we utilize SCM alongside DRAM, while still reducing the cost by 43-48% compared to large DRAM designs. To our knowledge, this is the first paper that demonstrates practical cost-performance trade-offs for potential deployment of DCPMM in commercial datacenters.

Acknowledgments

We like to thank our shepherd, Tudor David, and the anonymous reviewers for their thorough comments that greatly improved this work. We are also grateful to Yanqin Jin, Jesse Koh, Darryl Gardner, Siying Dong, Pallab Bhattacharya, Shumin Zhao, and many others at Facebook for their support and suggestions in this research project.

References

- [1] Facebook. Introducing “yosemite”: the first open source modular chassis for high-powered microserver. 2015. <https://engineering.fb.com/core-data/introducing-yosemite-the-first-open-source-modular-chassis-for-high-powered-microservers/>.
- [2] Facebook. Rocksdb. 2020. <https://rocksdb.org/>.
- [3] Intel. Intel® optane™ persistent memory. 2019. <https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html>.
- [4] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 24–33, New York, NY, USA, 2009. Association for Computing Machinery.
- [5] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 421–432, 2013.
- [6] Ju-Young Jung and Sangyeun Cho. Memorage: Emerging persistent ram based malleable main memory and storage architecture. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, page 115–126, New York, NY, USA, 2013. Association for Computing Machinery.
- [7] Yonatan Gottesman, Joel Nider, Ronen Kat, Yaron Weinsberg, and Michael Factor. Using storage class memory efficiently for an in-memory database. In *Proceedings of the 9th ACM International on Systems and Storage Conference*, SYSTOR '16, New York, NY, USA, 2016. Association for Computing Machinery.
- [8] J. Jeong, J. Hong, S. Maeng, C. Jung, and Y. Kwon. Unbounded hardware transactional memory for a hybrid dram/nvm memory system. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 525–538, 2020.
- [9] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. *SIGARCH Comput. Archit. News*, 39(1):91–104, March 2011.
- [10] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 105–118, New York, NY, USA, 2011. Association for Computing Machinery.
- [11] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 897–912, Renton, WA, July 2019. USENIX Association.
- [12] 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 '09, page 133–146, New York, NY, USA, 2009. Association for Computing Machinery.
- [13] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.
- [14] 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 Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.
- [15] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman 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, 2019.
- [16] Mikhail Zarubin, Patrick Damme, Dirk Habich, and Wolfgang Lehner. Polymorphic compressed replication of columnar data in scale-up hybrid memory systems. In *Proceedings of the 13th ACM International Systems and Storage Conference*, SYSTOR '20, page 98–110, New York, NY, USA, 2020. Association for Computing Machinery.
- [17] Kai Wu, Frank Ober, Shari Hamlin, and Dong Li. Early evaluation of intel optane non-volatile memory with hpc i/o workloads, 2017.
- [18] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

- [19] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single machine graph analytics on massive datasets using intel optane dc persistent memory. *Proceedings of the VLDB Endowment*, 13(10):1304–1318, Jun 2020.
- [20] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. Performance characterization of a dram-nvm hybrid memory architecture for hpc applications using intel optane dc persistent memory modules. In *Proceedings of the International Symposium on Memory Systems, MEMSYS '19*, page 288–303, New York, NY, USA, 2019. Association for Computing Machinery.
- [21] Ivy Peng, Kai Wu, Jie Ren, Dong Li, and Maya Gokhale. Demystifying the performance of hpc scientific applications on nvm-based memory systems. *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2020.
- [22] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. Asymnvm: An efficient framework for implementing persistent data structures on asymmetric nvm architecture. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20*, page 757–773, New York, NY, USA, 2020. Association for Computing Machinery.
- [23] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and availability via client-local NVM in a distributed file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1011–1027. USENIX Association, November 2020.
- [24] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 496–508, 2020.
- [25] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12*, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.
- [26] 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.
- [27] 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.
- [28] 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, Santa Clara, CA, February 2020. USENIX Association.
- [29] Facebook. db_bench. 2020. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools#db_bench.
- [30] Google. Leveldb. 2011. <https://dbdb.io/db/leveldb>.
- [31] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, 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.
- [32] Redis. Redis. 2020. <https://redis.io/>.
- [33] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.
- [34] Facebook. Rocksdb users and use cases. 2020. <https://github.com/facebook/rocksdb/wiki/RocksDB-Users-and-Use-Cases>.
- [35] Facebook. Rocksdb trace, replay, analyzer, and workload generation. 2020. <https://github.com/facebook/rocksdb/wiki/RocksDB-Trace%2C-Replay%2C-Analyzer%2C-and-Workload-Generation>.
- [36] Micron. 3d xpoint technology. 2020. <https://www.micron.com/products/advanced-solutions/3d-xpoint-technology>.
- [37] Intel. 3d xpoint™: A breakthrough in non-volatile memory technology. 2020.

<https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html>.

- [38] Intel. Intel optane dc persistent memory. 2020. <https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-Quick-Start-Guide.pdf>.
- [39] IgorsIstocniks. How whatsapp moved 1.5b users across datacenters. 2019. <https://docplayer.net/161220289-How-whatsapp-moved-1-5b-users-across-data-senters-igors-istocniks-code-beam-sf-2019.html>.
- [40] Muthu Annamalai. Zippydb: a modern, distributed key-value data store. 2015. <https://www.youtube.com/watch?v=DfiN7pG0D0k>.
- [41] Satadru Pan, Theano Stavrinou, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231. USENIX Association, February 2021.
- [42] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 383–398, Broomfield, CO, October 2014. USENIX Association.
- [43] Guoqiang Jerry Chen, Janet L Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. Real-time data processing at facebook. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1087–1098, 2016.
- [44] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruva Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1013–1020, 2010.
- [45] Facebook. Hive - a petabyte scale data warehouse using hadoop. 2009. <https://www.facebook.com/notes/facebook-engineering/hive-a-petabyte-scale-data-warehouse-using-hadoop/89508453919/>.
- [46] CK Chow. On optimization of storage hierarchies. *IBM Journal of Research and Development*, 18(3):194–203, 1974.
- [47] Intel. Ipmctl. 2020. <https://github.com/intel/ipmctl>.
- [48] Ndctl and daxctl. 2020. <https://github.com/pmem/ndctl>.
- [49] memkind library. 2020. <https://github.com/memkind/memkind>.
- [50] Intel. Intel server board s2600wftr specification. 2019. <https://ark.intel.com/content/www/us/en/ark/products/192581/intel-server-board-s2600wftr.html>.
- [51] Intel. Intel xeon gold 6252 processor specification. 2019. <https://ark.intel.com/content/www/us/en/ark/products/192447/intel-xeon-gold-6252-processor-35-75m-cache-2-10-ghz.html>.
- [52] OCP. Ocp tioga pass 2s server design specification v1.1. 2018. <https://www.opencompute.org/documents/open-compute-project-fb-2s-server-tioga-pass-v1p1-1-pdf>.
- [53] OCP. Ocp twin lakes 1s server design specification v1. 2018. <https://www.opencompute.org/documents/facebook-twin-lakes-1s-server-design-specification>.
- [54] Hyperscalers. Rackgo x yosemite valley. 2019. <https://www.hyperscalers.com/Rackgo-X-Yosemite-Valley>.
- [55] Hyperscalers. Rackgo x leopard cave. 2019. https://www.hyperscalers.com/OCP-Hyperscale-Systems?product_id=194.
- [56] Jim Handy. Intel's optane dimm price model. 2019. <https://themoryguy.com/intels-optane-dimm-price-model/#more-2291>.
- [57] JPaul Alcorn. Intel optane dimm pricing. 2019. <https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html>.
- [58] MATLAB. Fit power series models using the fit function. 2020. <https://www.mathworks.com/help/curvefit/power.html>.
- [59] MATLAB. gpfitt: Generalized pareto parameter estimates. 2020. <https://www.mathworks.com/help/stats/gpfitt.html>.
- [60] MATLAB. Fit sine models using the fit function. 2020. <https://www.mathworks.com/help/curvefit/sum-of-sine.html>.

- [61] 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.
- [62] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 30(10):2223–2236, 2004.
- [63] Compute express link™: The breakthrough cpu-to-device interconnect. 2020. <https://www.computeexpresslink.org/>.
- [64] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. System evaluation of the intel optane byte-addressable nvm. *Proceedings of the International Symposium on Memory Systems*, Sep 2019.
- [65] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Persistent memory i/o primitives. *Proceedings of the 15th International Workshop on Data Management on New Hardware - DaMoN'19*, 2019.
- [66] Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. Bridging the latency gap between nvm and dram for latency-bound operations. In *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN'19*, New York, NY, USA, 2019. Association for Computing Machinery.
- [67] Anil Shanbhag, Nesime Tatbul, David Cohen, and Samuel Madden. Large-scale in-memory analytics on intel® optane™ dc persistent memory. In *Proceedings of the 16th International Workshop on Data Management on New Hardware, DaMoN '20*, New York, NY, USA, 2020. Association for Computing Machinery.
- [68] Yinjun Wu, Kwanghyun Park, Rathijit Sen, Brian Kroth, and Jaeyoung Do. Lessons learned from the early performance evaluation of intel optane dc persistent memory in dbms. *Proceedings of the 16th International Workshop on Data Management on New Hardware*, Jun 2020.
- [69] S. Imamura and E. Yoshida. Fairhym: Improving inter-process fairness on hybrid memory systems. In *2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6, 2020.
- [70] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, Santa Clara, CA, July 2017. USENIX Association.
- [71] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 967–979, Boston, MA, July 2018. USENIX Association.
- [72] Katelin A. Bailey, Peter Hornyack, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Exploring storage class memory with key value stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads, INFLOW '13*, New York, NY, USA, 2013. Association for Computing Machinery.
- [73] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16*, page 371–386, New York, NY, USA, 2016. Association for Computing Machinery.
- [74] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing, ICS '11*, page 85–95, New York, NY, USA, 2011. Association for Computing Machinery.
- [75] Z. Zhang, Y. Fu, and G. Hu. Dualstack: A high efficient dynamic page scheduling scheme in hybrid main memory. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*, pages 1–6, 2017.
- [76] S. Bock, B. R. Childers, R. Melhem, and D. Mossé. Concurrent migration of multiple pages in software-managed hybrid main memory. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 420–423, 2016.
- [77] Haikun Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He, Long Zheng, and Rentong Guo. Hardware/software cooperative caching for hybrid dram/nvm memory architectures. In *Proceedings of the International Conference on Supercomputing, ICS '17*, New York, NY, USA, 2017. Association for Computing Machinery.
- [78] L. Liu, S. Yang, L. Peng, and X. Li. Hierarchical hybrid memory management in os for tiered memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2223–2236, 2019.
- [79] K. Wu, J. Ren, and D. Li. Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 401–413, 2018.

- [80] Ahmad Hassan, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. Software-managed energy-efficient hybrid dram/nvm main memory. In *Proceedings of the 12th ACM International Conference on Computing Frontiers, CF '15*, New York, NY, USA, 2015. Association for Computing Machinery.
- [81] H. Chang, Y. Chang, T. Kuo, and H. Li. A light-weighted software-controlled cache for pcm-based main memory systems. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 22–29, 2015.
- [82] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu. Utility-based hybrid memory management. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 152–165, 2017.
- [83] G. Dhiman, R. Ayoub, and T. Rosing. PDRAM: A hybrid pram and dram main memory system. In *2009 46th ACM/IEEE Design Automation Conference*, pages 664–669, 2009.
- [84] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys '18*, New York, NY, USA, 2018. Association for Computing Machinery.

First Responder: Persistent Memory Simultaneously as High Performance Buffer Cache and Storage

Hyunsub Song Shean Kim J. Hyun Kim Ethan JH Park Sam H. Noh
UNIST (Ulsan National Institute of Science and Technology)

Abstract

Persistent Memory (PM) is a new media with favorable characteristics that can vastly improve storage I/O performance. While new PM based file systems have been developed to exploit PM, most work have not been successful in fully integrating PM media with traditional storage media such as SSDs and HDDs. We present First Responder (FR), a means to exploit the beneficial features of PM, while making use of modern and mature file systems such as Ext4 developed for traditional storage devices. Conceptually, FR is much like a buffer cache, but much more is involved such as maintaining consistency under failure and providing featherweight management overhead. FR brings about multiple benefits. First, we retain the maturity of existing file systems allowing deployment of FR at settings where traditional file systems are deployed. Second, traditional storage devices supported by these file systems can be used allowing easy integration of PM with traditional storage. Finally, FR allows in-order file system semantics at close to PM device latency. With experimental evaluations with the Intel DC PMM, we show that FR, when used in cache form, can outperform Ext4 by more than 9×, while providing durable in-order file system semantics, whereas Ext4 cannot. We also show that when used as part of a typical file system, performance is comparable with NOVA and Ext4-DAX.

1 Introduction

Persistent Memory (PM) is a new media with favorable characteristics such as nonvolatility and close to DRAM performance that can vastly improve storage I/O performance. To exploit these characteristics, new PM based file systems are being developed [7, 8, 19, 23, 52, 58]. These file systems, while providing high performance, suffer from the following two limitations. First, as the time to maturity for a file system is long and a file system is a complex beast [11, 30, 42], most of these relatively young and immature file systems will not survive the test of time. For example, among the many file systems proposed, currently, NOVA [52] and DAX [36] are the only ones that are stable and being maintained. Second, how

these file systems will integrate with existing storage devices such as SSDs and HDDs is not straightforward. For example, both NOVA and DAX assume that the storage media is PM. In the long run, PM will fill up and some form of migration may be needed to vacate PM, which some file systems are not designed to do. Efforts to resolve this have been proposed. In their seminal paper, Kwon et al. propose a cross-media file system called Strata that can span a set of heterogeneous storage devices such as PM and traditional devices [23]. Separately, Ziggurat, proposed by Zheng et al., is a tiered file system that supports a combination of devices [58]. However, both proposals suffer from the first limitation; Strata, at the time of this writing, still is not fully functional [46] and Ziggurat is not open sourced and thus, its status is unknown.

The goal of this study is to encompass the goals of previous PM file system developments. More specifically, the goals of previously proposed PM based file systems can be summarized as follows. First, file system performance should extract the benefits that PM brings as storage media. This is an obvious goal when deploying PM. Second, it should support the more natural in-order file system semantics [23] brought upon by PM. In-order file system semantics is where all file system operations, including writes, occur in the exact order in which they are executed. Typically, this has been impossible due to the writes, which were too slow to persist in order because of the slow devices. Thus, application developers had to choose between performance and durability. While it is true that some applications do not require durability of writes or are highly optimized to minimize sync's, many applications knowingly choose performance over durability, being aware of the negative consequences of failures [10, 37, 42]. With the advent of PM, such choice seems unnecessary as providing in-order file system semantics should be natural and easy [23, 42, 52]. However, to date, supporting such semantics is possible only with a total revamp of the file system. Third, it should support traditional storage media alongside PM (similarly to Strata and Ziggurat) [23, 58]. Our work encompasses these goals and yet, instead of developing yet another file system, our design allows to retain the modern, mature, constantly evolving

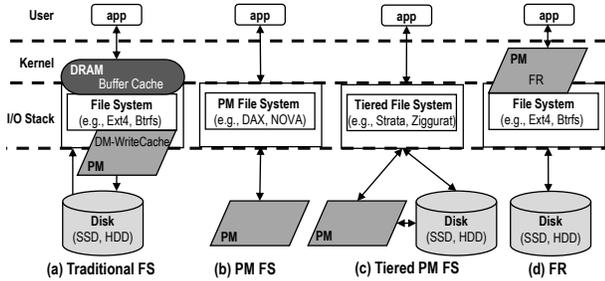


Figure 1: Comparison of PM use scenarios for file systems

file systems such as Ext4 developed for traditional storage devices.

In this study, we introduce First Responder (FR), the solution that we propose. FR, like a buffer cache, absorbs requests at the topmost layer of the I/O stack in PM, as shown in Figure 1(d), then immediately responds to the requests (hence, the name). For reads, the requests end there, while for writes, the requests are forwarded to the traditional file system, in effect, hiding the entire I/O stack overhead.

At first glance, FR simply appears to be nothing but a buffer cache that is persistent, but in reality, a lot more is involved. In fact, FR is a union of storage and a cache. As it will become more evident later, FR is storage as the contents that reside in FR are consistent and permanent like any that is stored in traditional storage media that have gone through the entire storage stack. Thus, in that sense, flushing to the traditional file system underneath is optional. FR is also a cache as the size is limited (though generally large) and eventually, it is flushed to the traditional storage media through the I/O stack. Naturally, the design of FR is much more involved as maintaining consistency under failure and providing featherweight management overhead become technical issues that need to be dealt with. We elaborate on these issues later.

There are two key benefits to FR. First, writes are immediately durable. Thus, even applications that do not require in-order semantics, but that issue occasional `fsync()` can benefit. Second, FR can coexist largely independently of the underlying traditional file system. This allows for any mature file system to take advantage of the benefits of PM. This is especially important as efforts to move to disaggregation of resources is gaining traction [3, 43, 48]. Thus, much extra effort to integrate PM based file systems to any conventional setting including disaggregated settings is not necessary with FR. While we concentrate on the Ext4 file system in most of our discussions, we also show performance results when using Btrfs [5, 41] as the underlying file system.

Our performance evaluations with the Intel DC PMM using an FIO generated synthetic workload show that FR, when used in cache form, can outperform Ext4 by more than 9 \times , despite providing durable in-order file system semantics, while Ext4 cannot. Using the Filebench and YCSB benchmarks, we also show that when FR is used as part of a typical file system, performance is comparable with the default Ext4, Ext4 with DM-WriteCache, NOVA, and DAX, while, again, providing

Table 1: Characteristics of file systems in Figure 1. In-order: In-order semantics support, Media: Storage media supported, Mature: Mature file system support. (* refers to underlying file system being mature, not that FR is.)

	(a) Trad. FS	(b) PM FS	(c) Tiered FS	(d) FR
In-order	No	Yes	Yes	Yes
Media	All	PM only	All	All
Mature	Yes	No	No	Yes*

durable in-order file system semantics.

In the remainder of the paper, we first discuss progress on work related to PM and distinguish how FR is different from them. Then, we present the design of FR in two separate sections; Section 3 discusses the overall design and in Section 4, we present a detailed discussion of how FR maintains consistency as this is key in providing a correct, yet efficient system to users. In Section 5, we perform a comprehensive evaluation of FR using the Intel DC PMM platform. We compare FR with NOVA and DAX, the two open source PM supporting file systems, and the default Ext4 and DM-WriteCache, which do not provide immediate durable in-order semantics supported by PM file systems. Then, we conclude in Section 6.

2 Related Work

Persistent Memory (PM) technologies represented by PCM (Phase Change Memory) [38] and STT-MRAM [22] are being considered as high performance storage mediums as they are nonvolatile and yet, provide random byte addressability and latency similar to DRAM. Intel recently commercialized the Optane DC PMM, the only product currently available in the market [15, 17, 53]. It can be used in one of various forms, specifically, as storage, as memory with DRAM as its cache, and as an extension of memory [14]. While many recent studies have considered file systems for PM [7, 8, 19, 23, 52, 58], our work is not about developing a new PM file system, but rather on integrating PM with existing modern file systems developed for traditional storage devices.

The closest set of related work are studies on the buffer (or page) cache. The buffer cache, as depicted in Figure 1(a), is a DRAM layer that attempts to hide the low performance of storage and has been a topic of study for generations, mostly concentrating on the replacement or prefetching policy issues [2, 18, 50]. This was important as performance between DRAM and storage was many orders of magnitude different. With the advent of higher performing storage devices, there have been arguments as to the need for the buffer cache [8, 21, 52] as well as attempts to revisit this topic [20, 24, 51]. In particular, the DM-WriteCache [20] is an interesting optimization that introduces a new nonvolatile layer, as PM or SSD, just before the slower storage device, which could be an SSD or HDD, as depicted in the lower part of Figure 1(a). DM-WriteCache, supported from Linux 4.18 and beyond, is a writeback cache that helps to improve performance by caching writes from the page cache to the storage media. However, as writes are first written to the page cache,

in-order file system semantics is not supported. The contrast between Figures 1(a) and (d) show how DM-WriteCache and FR are different by design.

There are also studies of the buffer cache that exploit the nonvolatile nature of PM [25, 40, 51]. UBJ, one of the first studies in this realm, makes use of PM based main memory as a buffer cache that unifies the functionalities of journaling and caching [25]. Tinca is a PM based buffer cache located in the external disk cache below the DRAM-based main memory [51] that uses a similar technique proposed in UBJ. FR is different from these studies in that durability is provided from the earliest layer of the I/O stack allowing immediate response. This requires careful design for consistency with the underlying file system, which these earlier studies neglect.

Retaining consistency is a key design issue for FR. Similarly, consistency has been the central issue in the design of numerous studies on data structures for PM. With efforts to reduce instructions that control write order [13, 35] such as `clwb` and `sfence` as these instructions incur considerable overhead [17], while, at the same time, abiding to the 8-byte failure atomicity restriction to maintain consistency, various data structures for PM have been proposed [12, 27, 31, 55]. FR conforms with these efforts as it carefully designs the use of 8-byte writes with `clwb` and `sfence`.

Other related studies include PM based file systems. Recent studies have proposed many optimizations including user-level approaches to avoid crossing the kernel boundary in an effort to improve performance [7, 19]. While so, only NOVA and DAX are stably supported in Linux. They all, more or less, share the common layout shown in Figure 1(b). Out of these, as mentioned previously, Strata and Ziggurat, are unique in their support of traditional storage media in the form depicted in Figure 1(c). These diagrams contrast the differences between them and FR. Finally, a recent study presents Assise, a new distributed file system based on Strata, where PM is used as a client-local cache layer [4]. The use of PM cache and its effectiveness is coordinated in a distributed setting through replication and choices between pessimistic and optimistic consistency modes. This is different from our approach where we exploit existing file systems and consistency is always immediate, though locally ensured. Table 1 summarizes how previous local file systems differ from FR.

3 First Responder: The Design

In this section, we present the design of First Responder (FR). Before moving on, we set the premise on which our discussion is presented. First, even though FR could be implemented in either the user or kernel layer, we will describe it within the latter layer as we implement it in the VFS layer. Second, as the main dealings of FR are with the `read()` and `write()` calls and as `read()` follows trivially from `write()`, we concentrate on `write()` in the following discussions. We first discuss the assumptions, then the design choices that we make based on the challenges we face.

3.1 Basic Architecture and Design Choices

Like many previous studies that assume memory to be a hybrid of traditional DRAM and PM, FR works under this assumption [4, 33, 52]. PM used by FR is a temporary storing ground, and yet it is also storage. That is, what is stored in PM is durable and consistent, though eventually, we anticipate its contents to be stored in traditional storage media such as SSDs and HDDs.

Traditional systems support two types of write modes, namely, synchronous and asynchronous. In FR, all aligned full chunk and/or new writes are synchronous, that is, they are persisted immediately into PM by FR. Writes of partial chunks already residing in underlying storage must first be read into PM for consistency, but we find them to be only a small portion of writes. FR manages the data and the location of where the data should be placed within PM, guaranteeing consistent and durable writes as PM is nonvolatile. Thus, applications upon receiving the response can continue execution being assured that the write I/O request was serviced successfully. Note that this has the consequence of dramatically reducing the I/O path for such synchronous writes compared to traditional file systems.

In the previous section, we mentioned that FR is like a buffer cache, but that much more is involved. We now discuss the two specific issues that must be resolved, that is, maintaining consistency upon failure and providing feather-weight management overhead, which the traditional buffer cache cannot support.

Handling Overhead: Recall that the traditional buffer cache was developed as an optimization to alleviate the burden due to slow storage, which was orders of magnitude slower than DRAM. Thus, the buffer cache maintained separate data structures and elaborate replacement algorithms were developed. Compared to accessing slow disks, maintaining such data structures and running these algorithms were negligible. However, as previously observed, software overhead considered to be negligible in the past is no longer so with faster media [26, 45, 54]. This becomes more important as PM is close to DRAM performance, which means that every little overhead counts. Thus, management with such high cost is unacceptable with PM systems.

To alleviate such overhead, FR chooses a simple, static indexing scheme for placement as well as replacement as we explain below. However, we must first argue that this is a valid approach. For this, we perform our own set of experiments to quantify the software effects of managing a cache, which we elaborate on in Section 5.2. Briefly, we implement indexing structures such as the Radix tree and the LRU replacement policy and find the overhead to be an order of magnitude higher than the static approach that we propose, concluding that with faster media such as PM, software overhead is profound.

The obvious question here, then, is, won't this static indexing increase the miss ratio? The obvious answer, of course, is yes; but we argue that this is true only under the traditional

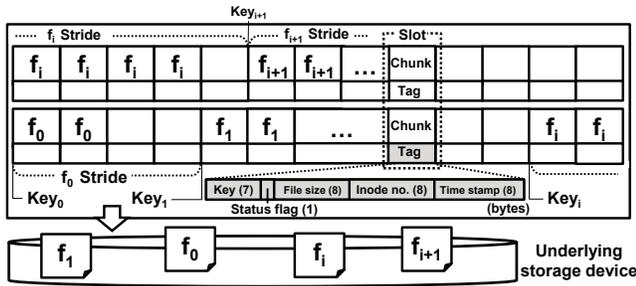


Figure 2: Internal organization and components of FR

assumption that the cache size is a very small fraction of storage capacity. This leads to our more important argument that, if the cache is sufficiently large, then, we may be able to avoid most cache misses. For reads, a miss is inevitable on the first read from storage. However, aside from these, it is well known that as the cache grows, the effect of the caching policy diminishes. For writes, a miss can be defined as a forced storage access when the new write collides with a block that is dirty. Then, so long as we keep all blocks clean, that is, synchronized with storage, then all misses can be eliminated. We show below that, under simplifying assumptions, if the cache is large enough all write misses can be avoided.

Imagine the cache to be organized in array form with the index i and S_b denoting the location and block size of the cache, respectively. Let C_s be the cache size and assume that for our system, the time to flush a block to storage and receive an acknowledgement (in order to synchronize) takes at most t_f time. Let us assume that the peak write rate for the cache media is r_{peak} GB/s. Assume that a very large write that is purely sequential arrives consecutively in S_b units and in t_i intervals, being placed from index $i = 0$ of the cache. To avoid all misses, the cache should be large enough such that when the entire cache fills up and wraps around to the beginning of the cache, the cache entries there are all clean. Thus,

$$t_f \leq \frac{C_s}{r_{peak}} \leq \frac{C_s \times t_i}{S_b} \Rightarrow C_s \geq \frac{t_f}{t_i} \times S_b \quad (1)$$

Then, with C_s satisfying Equation 1, no forced flushing of dirty blocks will occur for sequential writes.

Let us apply Equation 1 on an example. Assuming $S_b = 4$ KB, for current PM, we find that the peak write rate is roughly 4GB/s (not including any software overhead) when `clwb` and `sfence` operations are issued per chunk write. Thus, even when writes are coming in at full throttle, simple calculations show that a 128GB PM will need 32 seconds to wrap around the cache. In a recent article, Alibaba reported that it saw a peak order rate of 544,000/s [47]. While we understand that orders do not translate to actual I/O and, as Vuppapalati et al. report for the Snowflake infrastructure running on the Google Cloud Platform and Microsoft Azure [49], I/O requests come at more moderate rates, let us take the Alibaba numbers and say that an $S_b = 4$ KB request comes in every 2μ s, that is, $t_i = 2\mu$ s. Then, to wrap around the 128GB of PM, it takes 64 seconds.

In real life, of course, not all writes are sequential and other factors come into play. Also, keeping a cache in the GB range was not considered the norm though this is changing. One example is the Memory Mode use of PM where all of DRAM (which is in the many GB range) is used as a cache of PM [14]. However, as we will see later, sequential allocation of files is an important strategy in FR. Furthermore, as FR is based on the premise that PM is in the size range used by PM based file systems such as NOVA and DAX, for example, at least 128GB, this sufficiently large cache size allows the accommodation of a static indexing scheme, which we now describe.

Figure 2 shows, at a high level, how FR manages its space. FR uses a simple indexing structure based on a key associated with each file. (The details of what comprises chunks and slots as well as the how the indexing structures are maintained are given in Section 4.) Specifically, the index is simply calculated using $key_{c+1} = key_c + stride$ where key_c is the current key (initially 0) and $stride$ is a value dynamically determined for the workload. (Details of terms and their use are discussed in Section 4). In short, every file has its destined location within FR with no PM allocator involved, resulting in very little overhead for placement. Similarly, there is no need for replacement victim selection.

Handling Consistency: FR is a management module for data that resides in PM. That is, when a user requests a write, FR is the module that provides the response. The writes are made to PM, and they must be made durable such that they are recoverable upon system failure. For example, we cannot have torn writes; upon recovery from failure, its state should be either before the write or after the write. Also, as FR is the first responder to the underlying file system, eventually all data (and metadata) writes must reach the underlying storage, be it SSD or HDD. For example, consider a case where an overwrite of a block A, which resides in traditional storage happens. Upon a write, the user could receive an acknowledgement immediately after the data is placed in PM. However, this data and block A in storage are not in sync. Ensuring immediate in-order file semantics so that what the user perceives as valid data is indeed the (eventual) actual data, even in case of failures, is not a trivial matter. This may incur considerable performance overhead or require a completely new design of a PM-aware data structure, which is not an easy task as exemplified by previous studies [12, 27, 31, 59].

For this consistency issue, we develop a protocol under the PM 8-byte failure atomicity assumption. Like previous methods, the basic idea is to keep a dual copy of the data [16, 32], but the devil is in the details. Thus, a detailed and lengthy discussion is required, which we continue in the next section, along with a discussion on the failure recovery mechanism.

4 Data Consistency and Recovery Protocol

In this study, we say data in a file is consistent if, upon a write request to that file by an application, the data in the file is as before the write request or reflects all of the contents of

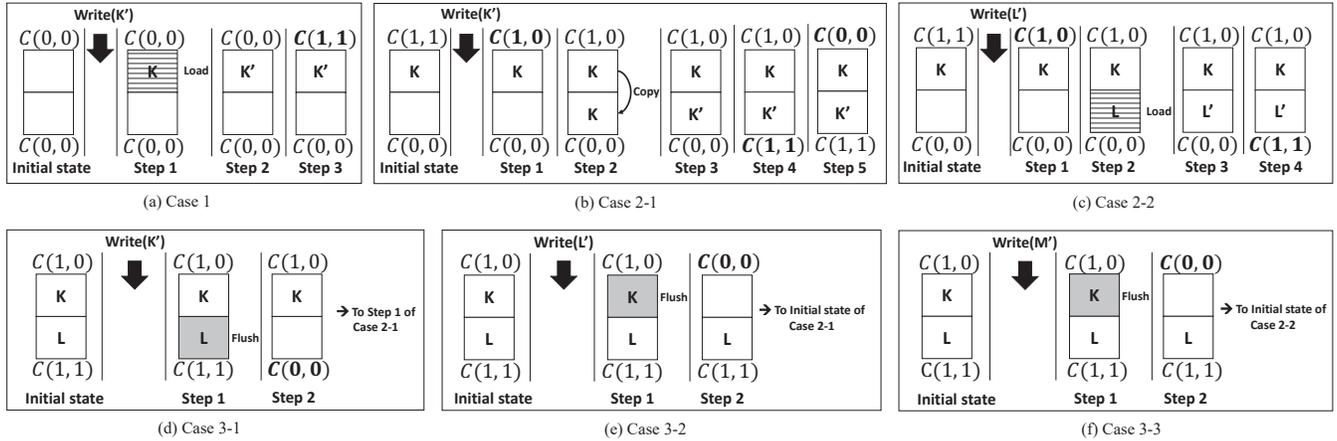


Figure 3: The figure shows the steps taken based on the initial condition of a slot when a write is requested. The $C(V,N)$ values on top and below the slot, that is, the two chunks are the status flags associated with each chunk. The changes to these status flags, denoted by the bold characters, are done atomically.

the issued write. Furthermore, guaranteeing data consistency means that write requests that have been acknowledged as having been written are eventually made durable and found through the underlying file system. In this section, we present the protocol that FR uses to guarantee data consistency.

4.1 The Basic Components

Figure 2 also shows the inner components of FR. There are a total of P continuous *chunks*, where a chunk is the load/store unit to storage, and P *tags*, with each tag being associated with one chunk. The chunks are organized in pairs, which we call a *slot*. As will be evident later, both chunks within a slot are used in full, making full use of the entire PM. The tag consists of the *key*, bits to represent the status of the corresponding chunks, the file size, the inode number, and the timestamp. The key is a unique number associated with the file slot assigned to it by FR upon file creation and stored in the inode of the file. This key is simply a global key maintained by FR, initialized to 0 and monotonically increased in *stride* increments upon every file creation, where the stride is set to reduce collisions, where a *collision* is defined as an allocation of two (or more) files to the same slot. We discuss how the stride is set in Section 4.3. Thus, the number of files that can be created is limited by the number of bits used to represent this value. In our current implementation, we use 56 bits and, for now, we make no effort to remedy this limitation as we think this is enough to represent the number of files for a system lifetime. There are other metadata in the tag, but they are discussed as the need arises.

4.2 Data Consistency Protocol

With PM, design for consistency must be done with the 8-byte failure-atomic write supported by the architecture. With traditional systems, providing durability with consistency becomes complex as the critical path to durability can be quite long.

However, as we elaborate below, with FR, no allocation algorithm is invoked and only a few bytes are needed to guarantee data consistency of the file system.

Writes: With FR, it is vital that writes to chunks are done in an atomic manner, and thus, immediately durable. We now present the core design of FR that quickly enables durable in-order semantics of the data written to the file system.

For our discussion, we assume that the key and status flag in the tag are written with 8-byte failure atomicity guarantee supported by the hardware, where the status flag contains 2 flags V and N (and other information that we ignore for now). V distinguishes valid versus invalid and N with value 1 denotes the new, more recently written chunk in the slot. We denote the flag states of chunk C as $C(V,N)$, and further, denote the states of the pair of chunks in slot S as $S[C_1,C_2]$. We regard $S[C_1,C_2]$ and $S[C_2,C_1]$ the same as the algorithm works in the same way given the two states.

The series of figures in Figure 3 show how an incoming write is managed. Upon a chunk write, the slot to place the chunk is determined by $(key \bmod \lfloor \frac{P}{2} \rfloor)$, which designates the first chunk of the file determined by the key attribute *key* assigned to each file upon creation, and its displacement from the start of the file. The write that is serviced by FR is denoted $Write(X)$, and X is an object that is of any size, though in our discussion, for simplicity, we assume that X is smaller than or equal to the chunk size. (If X is larger, then it simply takes multiple chunks, which we discuss separately in Section 4.4.) The pair of chunks in the slot indexed can be in one of three states, denoted above and below the chunks in Figure 3:

- $S[C(0,0),C(0,0)]$, the slot is empty, that is, both chunks of the pair contain no data (empty/invalid)
- $S[C(1,1),C(0,0)]$, only one chunk has valid data
- $S[C(1,1),C(1,0)]$, both chunks have valid data.

We now discuss each of the cases in detail. **Case 1:** This is when both chunks of the slot are invalid, that is, $S[C(0,0),C(0,0)]$. We can choose any chunk of the

pair to write to. Upon `write(K')`, if a chunk of the unmodified `K` exists in storage, it is first read in to the chunk.¹ Then, `K'` is written to the chunk and persisted with the `clwb` and `sfence` operations. Then, the tag `C(0,0)` is set to `C(1,1)`, and only after this setting is the request acknowledged.

Note that `clwb` and `sfence` operations must be performed by default for writes to FR for the write to persist. Hence, we assume that they follow by default unless otherwise mentioned and will be omitted in discussions hereafter. We reiterate that, here and throughout our discussion, these flags are part of the tag and changes to them are done atomically. The state of the chunk pair in FR before and after `write(K')` is shown in Figure 3(a).

Case 2: Here one chunk holds a valid object `K` while the other is invalid, that is, $S[C(1,1), C(0,0)]$. There are two cases to consider here; first is when `K` is being modified and the other is when a new object is being written, which can be distinguished by the uniqueness of the key. When `K` is updated, denoted by `write(K')`, `C(1,1)` is first modified to `C(1,0)`, stating that `K` is still valid but is no longer the new chunk. Then, `K` is copied to the chunk with state `C(0,0)` and the update is reflected in this chunk. Then, `C(0,0)` is modified to `C(1,1)`, and finally, `C(1,0)` is changed to `C(0,0)` to reflect that this chunk is now invalid. Thereafter, `write(K')` is acknowledged. This case is depicted as Case 2-1 in Figure 3(b).

Note that `K'` cannot be written to the first chunk directly, but must be overwritten on a copy of the chunk containing `K`. This is because the write may be an overwrite and failure during this write may result in a torn write. Also note that this case covers consecutive writes to the same chunk of the same file, and that in such cases all writes are being handled within the PM without any interaction with the storage device.

The second case within Case 2 is when a new object `L'` is written to the same slot as the chunk containing object `K` as in Figure 3(c). This will happen only when a different file is allocated to the same slot, that is, upon collision. Starting from the same initial state, `C(1,1)` is first modified to `C(1,0)` as this chunk will no longer be the new chunk in this slot. If `L'` is an overwrite of a chunk in storage, then the chunk containing `L` must first be read into the chunk with state `C(0,0)`, unto which object `L'` is written. Finally, `C(0,0)` is modified to `C(1,1)` as object `L'` is now valid and new.

Case 3: The final case is when both chunks in the slot are valid, that is, $S[C(1,1), C(1,0)]$, and a different valid chunk needs to be put into this slot. The initial states of such cases are shown in Figures 3(d)~(f). Let us assume the initial situation in these figures, where chunks containing `K` and `L` occupy the slot, with `L` being the newer of the two chunks. Note that this situation can arise only when a collision has already occurred as `K` and `L` must be of different files. Let us also assume for now that both `K` and `L` are dirty. There are three cases to

¹Note that if the write size is equal to the chunk size, then it need not be read from storage. Here, we are considering the worst case. This holds for subsequent discussions as well.

consider. The first two are updates of an existing object `K` or `L`. In these cases, the chunk containing the object that is not being updated is first flushed. This is to synchronize the dirty chunk with that in storage, which has the effect of making the chunk clean. Figures 3(d) and (e) depict these cases.

For Case 3-1 (and Case 3-2) where the older chunk `K` (the newer chunk `L`) is being overwritten, the other chunk `L` (`K`) is flushed and awaits the acknowledgement that the flush has completed. Once it arrives, `C(1,1)` (`C(1,0)`) is changed to `C(0,0)`, which results in a state identical to the initial state of Case 2-1. Thus, from here, the steps are the same as Case 2-1. Note that all flushes in FR are done by writing in Direct I/O mode (to be specific, with the `write_iter()` call).

Finally, the third case is when a new object is written to FR. This would be another collision with a new file on this slot. Case 3-3 in Figure 3(f) depicts this situation with `write(M')`. Similarly to the previous two cases, we must flush a chunk, and for this, we choose to flush the older chunk. Hence, in our example, the chunk containing `K` is flushed. Once the acknowledgement of the flush is received, `C(1,0)` is changed to `C(0,0)`. This results in the same initial state as Case 2-2. Thus, we proceed in the same steps as Case 2-2.

Note that, as described, Case 3 always incurs a flush to the storage device, which can be a source of overhead as the lock to the slot must be held until the acknowledgement for the flush is returned. Such flushes occur when a collision occurs and the chunk to be flushed is valid and dirty. However, note that if that chunk is either invalid or clean, the chunk need not be flushed. Fortunately, FR minimizes such flushes by design; the first source, that is, collision, is remedied by dynamically and judiciously assigning the stride value, while the second source, that is, dirty chunks, is remedied by periodically flushing the chunks in the background to make chunks clean, which we discuss in more detail in the next section. But first, we discuss how reads are handled.

Reads: Reads are similar to writes. First, the key is used to find the relevant slot. Then, a read hit is trivial as they are simply serviced out from FR. Upon a miss, we need to bring in data from storage. This case is similar to writes in that the invalid chunk will be used or if there is no invalid chunk, the old valid and dirty chunk will first be flushed out to a clean chunk. Then, the data requested will be brought in overwriting the clean chunk. The process of flushing, atomically changing the tag values, and loading follow similar steps as writes.

4.3 Strides, Periodic Flushing, and Metadata

Stride Setting: In FR, as a file is created, it is assigned a unique key, whose value is used to designate a fixed slot location in FR. Ideally, the files will be assigned as shown in Figure 2 with no overlap to avoid collisions. However, this is difficult to achieve as we need prior knowledge of the files that are created, even for those that may be appended to later on. If estimated to be too small, then collisions will occur, possibly incurring forced flushes. If estimated too large

(for example, f_0 in Figure 2), then internal fragmentation may occur leading to quicker wrap-around, for example, f_i in Figure 2. A wrap-around forces collision that could lead to forced flushes, leading to degraded performance.

We take an empirical approach in setting the stride. The basis of our approach is that file sizes are, in general, relatively homogeneous depending on the workload. For example, files in scientific workloads are small ranging in the few tens to hundreds of kilobytes [1, 28, 39], while files in database workloads are quite large being in the tens of MB to tens of GB range [29, 44]. Taking this observation, we create a Stride Table with $\langle \text{file size}, \text{file count} \rangle$ pairs where the ‘file size’ is a set of fixed numbers, and ‘file count’ is the number of files that are smaller but closest to ‘file size’. When a new file is created, we select the next larger ‘file size’ of the entry with highest ‘file count’ value as the stride value. The next larger one is selected as a larger value will lead to less possibility of collisions. As actual writes occur, the Stride Table is updated, possibly, decrementing one entry while incrementing another. Maintaining this table take only minimal memory, which in our implementation is 32KB.

Periodic Flush: While stride setting is one aspect of avoiding collision, periodic flushing is an active measure to avoid forced flushing due to collisions. Recall that forced flushing could have a negative effect on performance. To minimize this effect, we choose to periodically flush the dirty valid chunks to make them clean. Note that periodic flushing has no bearing on the consistency of data as FR is simultaneously storage and a cache. Thus, the period, p , can be set to 0 or to any value lower than the time to wrap around.

In selecting p , we take hints from Equation 1. For a sufficiently large C_s value such as 128GB or more, the t_f value may be sufficiently small to satisfy Equation 1 even under severely high request rates. Thus, a reasonable value that will not overload the underlying file system may be chosen. For our study, we choose to use $p = 10$ milliseconds as the default as this is the value that is generally used for the page cache.

Metadata: The discussion so far primarily focused on data and maintaining its consistency. As data is flushed to the underlying file system in periodic intervals, the metadata that is also maintained in the file system will only be synchronized within interval bounds. To remedy this, FR can take two measures. The first is regarding the time-related metadata such as `atime` or `mtime`. For this, FR keeps the actual timestamp for these metadata in the FR tag and flushes them as flushes occur, whether periodically or by forced flushes. The second measure is to modify metadata reading calls such as `stat()` to first flush the relevant chunks, and then, read the metadata. In FR, we have implemented the first measure, but only the `stat()` call for the second measure.

4.4 Multi-chunk Writes

While we have described writes for a single chunk, in reality writes may be composed of multiple chunks. We refer to such

writes as multi-chunk writes, which we now describe.

Fundamentally, multi-chunk writes are the same as single chunk writes. The key difference is that we provide a means to act upon the multi-chunks in an atomic manner. As accesses to chunks are controlled through individual locks, we acquire locks for each of the chunks in sequence from the first chunk to the last. Writes to chunks happen along with the acquiring of the locks. On writing the first chunk, the *not-complete* flag (another bit in the status flag byte in the tag) of this chunk is set to designate that the multi-chunk write has not yet completed. Then, the subsequent chunks are written to, along with the timestamp tag, where the timestamp of the first chunk is written. This timestamp plays a vital role in identifying all the chunks written from the same `write()` call when recovering from a failure. When all the chunks have been written to, the not-completed flag is reset, the write requesting application is notified, and the locks for all chunks are released, in this order.

4.5 Failure Recovery

The contents of FR are always recoverable from faults, be it hardware or software faults. The only exception is recovery from PM device failure where PM content is irrecoverable due to media failures or partial or full data corruption. Due to space, we only give the gist of recovery from single chunk write failure, which is that, for all faults occurring at any step in Figures 3(a)~(f), the recovered state will either be one of the initial states of the figures (in terms of the tags, specifically, $C(V,N)$ values of the chunks) or have already completed the intended write. For example, if a failure occurs during Step 1 of Case 3-2, that is, while waiting for the synchronous flush of chunk K to complete, when FR recovers, it will simply return to the initial state $S[C(1,0), C(1,1)]$ of Case 3-2.

Recovery from multi-chunk write failures is more involved, the key to the solution being the use of the not-complete bit to make sure recovery is done for all. However, we omit the detailed description as well as the experimental validation results, which we conducted, due to space.

5 Performance Evaluation

In this section, we evaluate the effects of FR on Ext4 performance, which we implement in Linux kernel version 4.18. A total of ~ 3000 lines of code were added; ~ 2900 for the FR module, ~ 70 in the VFS layer, and the rest in the Ext4 file system.

In the implementation, we modify five system calls, two of which are major and three are minor changes. In particular, minor changes are made to `creat()`, to assign the key, `fsync()`, to bypass the page cache, and `unlink()`, to clear the tag values once the chunks of the file to be deleted are found using the key. Major changes are made to `read()`, to make use of the key and Direct I/O in reading, and `write()`, to implement the consistency protocol, described in Section 4, with Direct I/O. One limitation of our current implementation of FR is that it does not support `mmap()`. This is because

Table 2: System configuration

	Description
CPU	Intel(R) Xeon(R) Gold 6242 CPU @ 2.80GHz (16 cores)
DRAM	Samsung 32GB 2666MHz DDR4 RDIMM×4 (128GB)
PM	Intel Optane DC Persistent Memory (512GB)
Storage	Samsung V-NAND SSD 860 EVO (1TB)
OS	Linux Ubuntu 18.04.3 LTS (64bit) kernel v4.18

`mmap()` is supported upon the page cache, but FR bypasses it. Thus, the semantics of a file simultaneously opened and accessed via `mmap()` and FR does not conform to POSIX. Resolving this issue is left for future work. The chunk size is set to 4KB and the 7-byte key, which contains the unique number assigned to the file upon file creation that is also stored in the inode, and the 1-byte status flag, which contains the V, N, the *dirty*, and not-complete bits, comprise the 8 bytes that are written in failure atomic manner.

5.1 Experiment Platform and Benchmarks

Table 2 lists the specifications of the basic experimental platform that we use. Our system is equipped with an Intel Xeon Gold 6242 CPU with 16 physical cores with 128GB DRAM and an underlying SATA SSD with 1TB capacity. The 512GB Intel Optane DC PMM is run on App Direct `fsdax` mode [14]. To format FR to the device and control the PM size used, we use `ioremap()`. Thus, all storage I/O occurs through this `ioremapped` FR region. For FR, as default, we make use of 128GB of the 512GB as storage, and periodic flushing occurs every 10ms as mentioned previously (denoted FR-10m or simply FR). We make use of a synthetic workload derived with the FIO tool as well as two sets of benchmarks that we describe in detail later.

5.2 PM as a Cache

In a typical storage setting, the underlying file system manages a large storage device. Also, typically, this storage is not in full use, filled with data only to some capacity, which we refer to as the dataset. Of this dataset, a fraction of it would be accessed at any given time, which will be the working set. This working set will grow and shrink as time evolves as files being accessed will come and go, and access intensities to them will also vary with time. The role of an effective cache is to hold the working set in the cache.

To evaluate FR as a cache on a general workload, we make use of synthetic workloads generated, in the following manner, using FIO [9], which is an I/O testing tool. Using FIO, we generate 21 distinct files using the characteristics shown in Table 3, whose dataset size is the aggregate of all the file sizes, that is, 154GB. The files themselves all have distinct access characteristics that are randomly selected from the bounds shown in the table. For example, one file will be 2GB in size, the file accessed in 8KB units with Pareto 0.1 distribution, with a read:write ratio of 95:5 of which 2% of the writes are synchronized, with a request interval of 1 microsecond, which

Table 3: Characteristics of the 21 files used to generate the synthetic workload. Numbers in brackets are number of files in the particular mix. N: Normal, R: Random.

	Description
File size	1~14GB [whole numbers only: 1 to 2 files of each]
Distribution	Pareto 0.1/0.5 [2/2], Zipf 0.2/1.2 [5/1], N [6], R [5]
IO unit	4KB [14], 8KB [7]
Read:Write	1:0 [4], 19:1 [9], 1:1 [8]
Fsync	0~5% [whole numbers only: 2 to 5 files of each]
Intensity	Request interval: 1μs~1ms (randomly distributed)

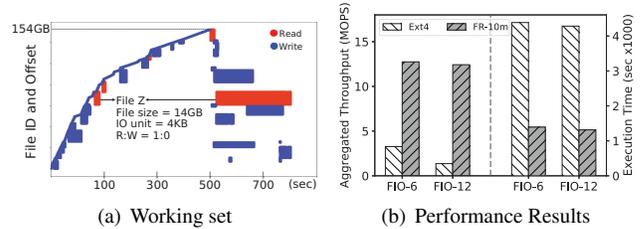


Figure 4: (a) Changing of the working set with time for the synthetic workload, only showing the first 900 of a 1300 second run. (b) Performance comparison between FR and Ext4 for light and heavy synthetic workloads.

is the request intensity of the Alibaba workload [47]. Note that the request interval of 1 millisecond would be the intensity of the Snowflake workload [49], so overall, the intensity is quite high. 21 files, each with a unique set of characteristics are generated. Then, we control the working set by controlling the number of files that are accessed in parallel. Figure 4(a) shows how the working set changes, where the y-axis represents the files accessed (by accessed offsets within the file), accumulated by the file access order from bottom to top, with time on the x-axis. The maximum number of files accessed in parallel at any point in time is set to 6, for light workload (FIO-6), and 12, for heavy workload (FIO-12), periodically starting new file accesses to meet this number. The working set of the light workload tops at roughly 50GB, while for the heavy workload, it is roughly 100GB. As all the 21 files finish off its accesses, the same 21 files are once again accessed repeatedly, but in random order starting another round of accesses. The second round of activities are shown to start just before the 500 second mark in the x-axis in Figure 4(a), and we observe that the access order is now changing. We also observe that the same files are being accessed much longer in the second round. For example, for the file marked File Z, which is a read-only file, the bar length, which represents time, is much longer on the second round than the first. (The height of the bar represents the file’s footprint.) This is because, it turns out, the number of files being accessed together is larger than at the first round, taking it longer to execute. While to the naked eye, the coloring of the bars look the same, when observed more closely, the longer right bar is considerably more sparse than that on the left meaning that it is taking longer to execute the same set of requests. To fully stress the

Table 4: Average latency for managing FR for various indexing and management policies. The average latency for FR static indexing is 67ns while for hashed indexing it is 75ns. (* Insert includes mechanism to find empty blocks.)

Activity		Radix-tree + LRU	Hash + LRU
Radix-tree / Hash	Hashing	-	78ns
	Search	35ns	162ns
	Insert*	19 μ s	19 μ s
	Delete	190ns	43ns
LRU	Touch	161ns	153ns
	Add	73ns	65ns
	Remove	88ns	82ns

system, we repeat this rounding two more times for a total of four rounds of accesses, randomly selecting the files to execute as time progresses. The total I/O for these experiments was around 575GB, and it is similar for both workloads as they are essentially making requests with the same characteristics. For FR the number of wrap-arounds took place about 19 times. To access individual files in parallel, each sequence of accesses to a file is generated by a separate thread run on a separate physical core. As the number of files accessed for these experiments is small, instead of the stride size to be that based on the most often seen file size, the size of the largest file seen is used. This has the negative effect of increasing the number of wrap-arounds, thereby increasing collisions.

Figure 4(b) compares the results of Ext4 and FR, the left bars showing the aggregated throughput of all the file accesses and the right bars showing how long the experiments took to finish. Note that we use the exact same file access sequence for the two cases, by monitoring the file access sequence when first executing with FR and then using that to execute Ext4. The results show that FR provides more than $9\times$ higher aggregate throughput and ended over $3\times$ faster than Ext4, despite the fact that FR is providing immediately durable in-order semantics. Note that write synchronization is light, with only relatively low write requests of which the sync rate is a maximum of 5%. We find that slight increase to sync rates to be bounded to 10% (which we believe is still light) diminishes Ext4 performance by roughly another 10-fold. Finally, we emphasize again that NOVA and DAX could not execute because the dataset was larger than the PM size.

PM Management Overhead: Another important aspect regarding the cache is its management. One could argue that instead of the rather radical static placement approach that FR uses, a more intelligent scheme based on sophisticated replacement algorithms could benefit FR. We briefly mentioned earlier that this would be too heavy when used with PM. Or, if it is going to use a static scheme, why not just use hashing? To check this, we implement multiple variations for managing PM such as hashing with LRU replacement, a file-based Radix tree just like that used in the Linux page cache along with the LRU replacement policy as well as just hashing for static indexing without a replacement policy. For the hash-

ing function, we found several open sourced ones and out of those used djb2 [57], which showed the best performance. All other code, except for the LRU policy, which was an in-house implementation, were taken from Linux. Table 4 shows the average overhead of the various components for managing the FR layer with the various schemes, when run with the Fileserver benchmark (whose characteristics are presented in the next section). We observe that the overhead can be orders of magnitude higher than our approach.

We conducted experiments using these data structures with the Fileserver workload and found that they were around $13\times$ slower compared to FR. (The benchmarks and the execution platform are described in the next section.) When simple hashing alone is used for placement, the overhead itself is low at 75ns, but we find that too many FR collisions (not hash collisions) occur incurring too much overhead for it to be a feasible approach.

5.3 Smaller than Cache Size Workloads

To compare FR performance with state-of-the-art PM file systems such as NOVA or DAX, we need to reduce the dataset size. In this section, we discuss the results for such a setting. Along with baseline Ext4, we also compare with Ext4 with PM DM-WriteCache (denoted DM-WC). However, it must be noted that DM-WC uses double the resources of the other schemes as it makes use of the entire DRAM as the page cache as well as 128GB of PM for the DM cache. Finally, we also attempted to compare with Assise [4]. Unfortunately, it would only run stably with a single thread and only for the workloads that the authors provided, that is, Fileserver and Varmail. As of this writing, other workloads did not execute properly even for single threaded execution. Multi-threading also was not supported. Thus, we do not include their results.

Benchmarks: As we can make use of traditional benchmarks in this setting, we make use of two benchmarks for our experiments whose characteristics are summarized in Table 5. The first is the Filebench benchmark, but of these we use only the Fileserver (F), Varmail (V), and OLTP (O) benchmarks. While it may be considered inappropriate to select particular workloads out of a benchmark suite, as FR supports in-order file system semantics these three benchmarks represent the extreme ends of the spectrum of workloads. Varmail and OLTP are workloads that have a considerable number of `fsync()` calls, while Fileserver is one that does not have any such calls.

The second benchmark is YCSB [6], which is generally used to evaluate NoSQL database systems, with the underlying database server being RocksDB. YCSB-A (YA) and YCSB-F (YF) are write heavy workloads (1:1 reads/writes), YCSB-B (YB) and YCSB-D (YD) are read intensive, YCSB-C (YC) is read-only, and YCSB-E (YE) is a range query workload. We follow the execution sequence recommended by the original YCSB authors [56] including the loading phases, but do not report the results for the load phases. All experiments are performed in async mode, thus data loss may occur for

Table 5: Characteristics of benchmarks. Request distribution for YCSB-D (YD) is Latest, while all others are Zipfian.

Filebench	R:W	Mean file size	# of files	Key-value store	Records Aggregated	R:W
Fileserver	1:2	128KB	200K	YA, YF	4GB	1:1
Varmail	1:1	32KB	800K	YB, YD, YE	4GB	19:1
OLTP	1:1	1.5GB	20	YC	4GB	1:0

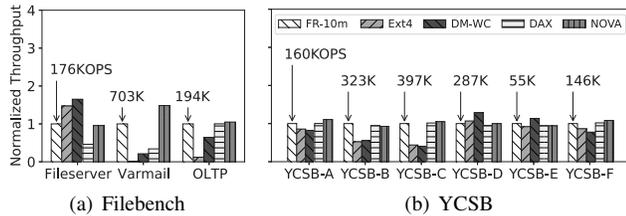


Figure 5: Filebench and YCSB throughput normalized to FR, with its absolute performance numbers shown.

Ext4 and DM-WC, while for DAX, NOVA, and FR, which provide in-order semantics, no loss will occur.

For all the workloads, we employ 16 threads as our system has 16 physical cores, except for OLTP, which employs 10 write threads and 20 read threads. For the Filebench workloads, the file size and number of files are set so that the dataset amounts to roughly 32GB. During execution, files are constantly being created and deleted, but overall the dataset does not deviate much from 32GB. For the YCSB workloads, we set the `recordcount` to 4 million, the `fieldcount` to 10, and the `fieldsize` to 100 bytes for records aggregated to 4GB. With these setting, the number of files created is around 450 and the average file size is roughly 80MB resulting in the dataset being in the 32GB vicinity. Our observations while these workloads are running on these datasets show that the maximum use of the page cache (for Ext4) at any time is roughly 10GB. Note that this setting allows the native file system to perform at its peak as the entire workload will fit into the page cache throughout its execution. All results reported are the averages of at least five runs each. We note beforehand that in multiple occasions, we only present partial results for particular discussions, unfortunately, due to space.

Filebench Performance: Figure 5(a) shows the performance results for the Filebench workloads. All results hereafter, unless otherwise mentioned, are shown relative to FR with the absolute values of FR indicated. For Fileserver, the results show that FR performs worse than Ext4 and DM-WC, while compared to DAX and NOVA, it does better. The reason Ext4 and DM-WC does better is that they provide the best environment for Fileserver as nearly all reads and writes are occurring in DRAM. This is because Fileserver does not make any `fsync()` calls (thus lacking durable in-order file system semantics). In contrast, for FR all access are at PM. PM read is roughly 3 times slower [17] and one-third of the workload are reads, and with the added peculiarities of PM [53], Ext4 and DM-WC should do better. In contrast to Fileserver, we see that for the Varmail and OLTP workloads, which are `fsync()`

Table 6: Various characteristics of workload execution. (M: million, WA: wrap-arounds)

	Flush/Access	Mem cpys	# of files	Average stride \times 4KB	Footprint (GB)	# of WA
F	1M/70M	35M	1M	614KB	572	9
V	0.6M/64M	52M	4M	410KB	1536	24
O	0/19M	22M	21	1.6GB	32	0.5
YA	100/25M	25M	450	268MB	112.5	2

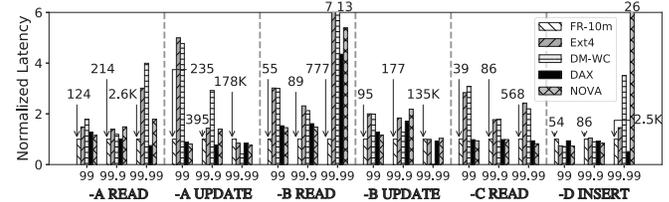


Figure 6: YCSB 99th, 99.9th, and 99.99th percentile tail latency as reported by YCSB normalized to FR. Numbers pointing to FR bars are absolute latency for FR in microseconds.

heavy, performance of FR is, respectively, roughly $94\times$ and $8\times$ better than Ext4 and roughly $4.5\times$ and $1.5\times$ better than DM-WC. Compared to NOVA and DAX, FR performance is slightly lower than NOVA, while DAX suffers for Varmail.

YCSB Performance: Figure 5(b) shows the results for YCSB. We observe that FR, NOVA, and DAX show similar performance, while Ext4 and DM-WC do the worst for the YCSB-A, YCSB-B, and YCSB-C workloads, while for YCSB-D, YCSB-E, and YCSB-F they are comparable or do better. However, we emphasize once more that neither Ext4 nor DM-WC guarantee in-order semantics. Note, in particular, the results for YCSB-C. As this is a read-only workload, one would expect Ext4 to perform similar to or better than FR as the entire dataset should be accessed from the page cache. However, we observe the contrary. This is because we run the workload with RocksDB, which issues around 50 `fsync()`s to manage a small number of files and perform compaction (movement between levels). In contrast, in terms of collisions in FR, YCSB-C is ideal as none occurs. Nuances of the effect of media are also observed when we compare the YCSB-C results from Figure 5(b) and Figures 9(a) and 9(b) that uses faster NVMe SSD and slower HDD media. We observe that the results for FR are the same for all three media, while for Ext4, performance is proportional to media performance, specifically, 436KOPS, 174KOPS, 153KOPS for NVMe SSD, SATA SSD, and HDD, respectively.

As YCSB also reports tail latency numbers we report them in Figure 6. The results show that Ext4 does worst in many cases. NOVA and FR are comparable but overall, FR does better, especially at the 99.99 tail.

Sources of Forced Flushing: We now analyze the sources of the performance differences. As FR works with large PM and the default periodic flushing is 10ms, one should expect FR to be free of any forced flushing. This is not what we observe.

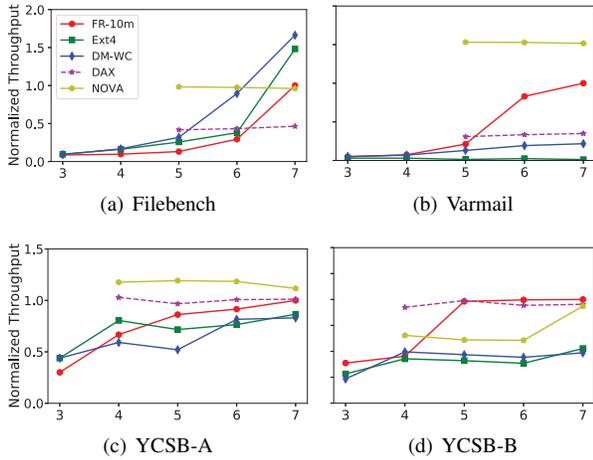


Figure 7: Performance results for PM size of 2^x GB, where x is value of points in x -axis, normalized to the performance of FR when $x = 7$.

To help understand, we make use of Table 6, which shows how the workloads execute.

The ‘Flush/Access’ column shows the total chunk accesses of the workloads of which ‘Flush’ are the forced flush counts, while the next column shows the total number of copies that occurred between chunks within slots. The next two columns show the number of files and average space allocated for the files as calculated by the stride multiplied by chunk size. These are different from those of Table 5, whose numbers are settings to run the workload, while here, they are those observed during execution, counting all files that were created. The final two columns show the footprint of the files within FR obtained by multiplying the values of the previous two columns and how many times the workload wrapped around, respectively, during the entire execution of the workload.

As observed from the ‘Flush/Access’ column, forced flushing occurs for around 1% of the accesses. There are two sources of forced flushes. The first is stride mis-estimation. While strides are set dynamically based on request size, it cannot foresee files that will grow. Hence, as files grow, overlap between neighboring files can occur, resulting in forced flushes. Detailed analysis (not shown) show that for the Filebench workloads, these situations are rare, but for YCSB-A, they account for all 100 flushes. The second source is wrap-around, as files are allocated in sequence in FR. We find that most of the forced flushes for Fileserver and Varmail are due to this. We observe from Table 6 that over a million files are being created, and the last two columns show that each slot is being shared by many files. Thus, collisions occur, possibly forcing flushes and resulting in performance loss.

5.4 Other Factors

Effect of PM Size: Figure 7 shows the performance of a select group of workloads (again, for clarity) when we vary the

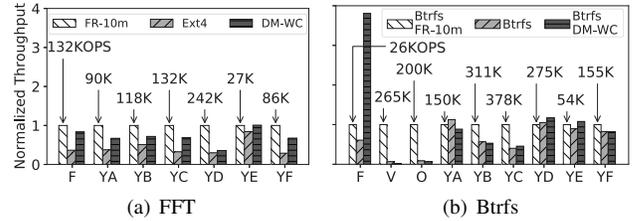


Figure 8: (a) YCSB executed with FFT modules taking up 108GB of DRAM capacity and (b) FR applied to Btrfs.

PM size, reducing it by half for each point to the left, for the various schemes. For DM-WC, both the DRAM and PM size is reduced by halves. All performance numbers are normalized to the 128GB FR results. There are two observations that can be made. First, the overall trend for FR, Ext4, and DM-WC are similar, though the drop in performance for FR is slightly greater. FR is performing similarly to the other two caching methods as the cache size gets tight. Second, for NOVA and DAX, there is barely any performance drop with the size drop. However, as PM drops beyond 32GB/16GB, they are not able to execute. This is the same situation as the synthetic workload experiments where the dataset exceeds the PM size at which point they are not able to execute. This, again, shows the limitations of PM based file systems.

Compensating for Extra PM: In our system configuration, PM is extra cost, but FR frees up DRAM used for the page cache. To evaluate this effect, we set up an environment where we have a memory intensive application, the FFT workload of Splash2x in the Parsec version 3.0 benchmark suite [34] set to use 12GB of memory, to run concurrently with the original workloads. Nine of these modules are invoked to take up 108GB of memory. The results are shown in Figure 8(a).

Overall, the results show that, while the absolute performance drops compared to those shown in Figure 5, the performance gap between FR over Ext4 and DM-WC grows. In particular, for Fileserver, we now see FR performing considerably better than both Ext4 and DM-WC. These results show that FR can effectively segregate I/O and memory intensive workloads relieving DRAM capacity for other use.

Btrfs: To show that FR is applicable to other file systems, we present results for FR with Btrfs [41], where changes are made to roughly ~ 30 LOC. As shown in Figure 8(b), the results are similar to FR-Ext4, with many cases performing better when FR is deployed, despite the fact that Btrfs is not providing durable in-order semantics. Notable differences compared to Figure 5 are that 1) FR-Btrfs does better even for Fileserver compared to Btrfs though in terms of absolute throughput, Btrfs does considerably worse than Ext4, 2) performance improvements with DM-WC are large, and 3) for YCSB-A and YCSB-D, Btrfs does better than FR-Btrfs.

Effect of Storage Media: We consider how FR is affected when the underlying storage media is changed to a higher-end Samsung 1TB V-NAND 970 PRO NVMe SSD and a

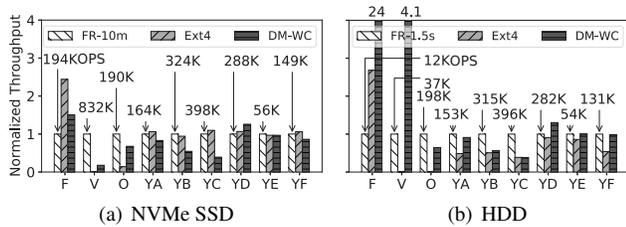


Figure 9: Performance with different storage devices.

lower-end Seagate 1TB Barracuda SATA3/7200/64M HDD. Figure 9(a) shows that with the NVMe SSD, the absolute throughput of FR improves benefiting more the Filebench workloads. More interestingly, we see that the stock Ext4 benefits considerably especially for the YCSB workloads. This is due to the bandwidth of the media. That is, FR is physically organized using a single PM chip with a throughput of 0.58GB/s, while the NVMe SSD throughput is 2.7GB/s. In some cases, Ext4 is able to exploit this, resulting in performance surpassing FR as exemplified by some of the YCSB results. Note again, however, that FR is providing immediate durability, while Ext4 and DM-WC are not, and that DM-WC uses double the resources. Note also that FR still shows superior performance for applications where synchronization is prevalent as shown by the Varmail and OLTP results. Another implication of these results is that we may not need to move to these more expensive devices if PM is deployed appropriately. The YCSB results shown in Figures 5 and 9 show that they are all similar irrespective of the underlying storage media be it an SSD or an HDD. However, the Filebench results forbid us from generalizing so easily. The reasoning behind these discrepancies are left for future work.

For the experiments with HDD, the periodic flushes for FR is set to 1.5 seconds to accommodate the slow HDDs. If set to the default 10ms, we find that performance drops considerably because the HDD cannot sustain service for such heavy I/O. (We have studied the sensitivity of period setting on performance, but omit them due to space.) The results show that 1) even with the large period, for Fileserver and Varmail, the request rate is too high for the HDD to hide and thus, we see drops in throughput from 194KOPS to 12KOPS and 832KOPS to 37KOPS for Fileserver and Varmail, respectively, when the media is changed from NVMe SSD to HDD, and 2) for other workloads, we see performance that is similar to that of FR when using the SATA SSD (Figure 5).

Let us now focus on DM-WC that takes on the best of both worlds by using DRAM as a read cache and PM for the write cache (though, using double the resources). We observe in Figure 9(b), with HDD, how DM-WC is almost always superior over Ext4 and even sometimes performing better than FR. Interestingly, as we move to a faster device (SATA SSD in Figure 5), the improvements diminish, sometimes even resulting in performance worse than Ext4. Then, with the fastest device (NVMe SSD in Figure 9(a)), we see the effect

of DM-WC diminishing further, with the majority performing worse than Ext4. There are two reasons for this degradation. One is device performance. DM-WC was devised to take advantage of superior device performance (PM, in this case), which is evident with the HDD results. However, for SATA SSD (0.52GB/s bandwidth), its performance is similar to that of PM (0.58GB/s). Thus, using PM should bring only a small improvement if any, meaning that, for example, for YCSB-A and YCSB-F, the write intensive ones in Figure 5, we should see comparable or slightly better performance for DM-WC compared to Ext4. However, we observe that DM-WC performs worse than Ext4. This is because of the second reason for performance degradation, that is, management overhead. DM-WC goes through an elaborate sequence of calls within (omitting the details, it goes through eight or so function calls). This incurs considerable overhead of around 35 microseconds of which the majority is overhead for indexing. This is pure overhead and is compounded as more writes are issued. Thus, write intensive workloads YCSB-A and YCSB-F are taking the blow. These results are evidence of the need for light management mechanisms such as that we propose. Finally, with the NVMe SSD, Ext4 performs better because the SSD has higher bandwidth (2.7GB/s vs 0.58GB/s). However, the added capacity does help DM-WC for some workloads.

6 Conclusion

In this paper, we presented First Responder (FR), a means to exploit the beneficial features of PM. While FR shares the goal of all PM based file systems in exploiting the performance benefits of PM as a storage device, its approach is unique in that it allows the use of existing modern file systems. Conceptually, FR is much like a buffer cache, but we showed that much more is involved as consistency had to be maintained under failure while providing light management. Built at the VFS layer, while the rest of the I/O stack, including the specific file system layer, remained largely unchanged, FR was able to provide immediate response to users from this layer. In experimental evaluations with the Intel DC PMM, we showed, with the FIO synthetic workload, FR, when used in cache form, can outperform Ext4 by more than $9\times$, while providing durable in-order file system semantics. Using the Filebench and YCSB benchmarks, we also showed that FR, when used as part of a typical file system, performs comparably with the default Ext4, Ext4 with DM-WriteCache, NOVA, and DAX, while also providing durable in-order semantics.

Acknowledgements

We thank our shepherd Professor Sasha Fedorova and the anonymous reviewers for their invaluable comments. We also thank Choulseung Hyun, Sunghwan Kim, Se Kwon Lee, and our colleagues from the NECSST lab for their numerous discussions that helped shape this research. This work was supported by Samsung Research Funding Centre of Samsung Electronics under Project Number SRFC-IT1402-52.

References

- [1] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A Five-Year Study of File-System Metadata. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2007.
- [2] Thomas Alexander and Gershon Kedem. Distributed Prefetch-buffer/cache Design for High Performance Memory Systems. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 1996.
- [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 *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2020.
- [4] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.
- [5] Btrfs. Btrfs. <https://github.com/torvalds/linux/tree/master/fs/btrfs>.
- [6] 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)*, 2010.
- [7] 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 ACM Symposium on Operating Systems Principles (SOSP)*, 2019.
- [8] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rahesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2014.
- [9] Freecode. fio. <http://freecode.com/projects/fio>.
- [10] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2018.
- [11] Jian Huang, Moinuddin K. Qureshi, and Karsten Schwan. An Evolutionary Study of Linux Memory Management for Fun and Profit. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2016.
- [12] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2018.
- [13] Intel. Intel 64 and IA-32 Architectures Software Developers Manual Combined Volumes. <https://software.intel.com/en-us/articles/intel-sdm>.
- [14] Intel. Intel Optane DC Persistent Memory Quick Start Guide. <https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-Quick-Start-Guide.pdf>.
- [15] Intel. Memory Optimized for Data-Centric Workloads. <https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html>.
- [16] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.
- [17] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman 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. Technical report, Computer Science Engineering, University of California, San Diego, 2019.
- [18] Song Jiang and Xiaodong Zhang. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the ACM Special Interest Group on Performance Evaluation (SIGMETRICS)*, 2002.
- [19] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, and Kolli Aasheesh. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.
- [20] Kernel Documentation. Writecache target. <https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/writecache.html>.

- [21] Hyojun Kim and Seongjun Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2008.
- [22] Emre Kultursay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.
- [23] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.
- [24] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2013.
- [25] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2013.
- [26] Eunji Lee, Hyokyung Bahn, Seunghoon Yoo, and Sam H. Noh. Empirical Study of NVM Storage: An Operating System’s Perspective and Implications. In *Proceedings of the IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2014.
- [27] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2017.
- [28] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2008.
- [29] Bunjamin Memishi, Raja Appuswamy, and Marcus Paradies. Cold Storage Data Archives: More Than Just a Bunch of Tapes. In *Proceedings of the 15th International Workshop on Data Management on New Hardware (DaMoN)*, 2019.
- [30] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnappalli, Pandian Raju, and Vijay Chidambaram. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.
- [31] 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)*, 2019.
- [32] Matheus Almeida Ogleari, Ethan L. Miller, and Jishen Zhao. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.
- [33] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A High Performance File System for Non-Volatile Main Memory. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2016.
- [34] PARSEC. PARSEC. <https://parsec.cs.princeton.edu/index.htm>.
- [35] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory Persistency. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2014.
- [36] Phoronix. XFS Will Get DAX Support In The Linux 4.2 Kernel. https://www.phoronix.com/scan.php?page=news_item&px=XFS-Linux-4.2-DAX-And-More.
- [37] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application crash consistency and performance with CCFS. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2017.
- [38] Simone Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, T. C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. H. Chen, H. L. Lung, and C. H. Lam. Phase-Change Random Access Memory: A Scalable Technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.
- [39] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

- [40] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2018.
- [41] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage (TOS)*, 2013.
- [42] Thanumalayan Sankaranarayanan Pillai, Vijay Chidambaram, Ramnathan Alagappan, Samer Al-Kiswany, and Andrea C. 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)*, 2014.
- [43] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiyang Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.
- [44] Juan Sillero and Javier Jiménez. Editorial opinion: public dissemination of raw turbulence data. *Journal of Physics: Conference Series (JPCS)*, April 2016.
- [45] Hyunsub Song, Young Je Moon, Se Kwon Lee, and Sam H. Noh. PMAL: Enabling Lightweight Adaptation of Legacy File Systems on Persistent Memory Systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017.
- [46] Strata. Strata: A Cross Media File System. <https://github.com/ut-osa/strata>.
- [47] TECHPP. Alibaba Singles' Day 2019 had a Record Peak Order Rate of 544,000 per Second. <https://techpp.com/2019/11/19/alibaba-singles-day-2019-record/>.
- [48] Shin-Yeh Tsai, Yizhou Shan, and Yiyang Zhang. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020.
- [49] 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 USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.
- [50] Carl A. Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache Modeling and Optimization using Miniature Simulations. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.
- [51] Qingsong Wei, Chundong Wang, Cheng Chen, Yechao Yang, Jun Yang, and Mingdi Xue. Transactional NVM Cache with High Performance and Crash Consistency. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017.
- [52] 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)*, 2017.
- [53] 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 USENIX Conference on File and Storage Technologies (FAST)*, 2020.
- [54] Jisoo Yang, Dave B. Minter, and Frank Hady. When Poll is Better Than Interrupt. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2012.
- [55] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, and Khai Leong Yong. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2015.
- [56] YCSB. Core Workloads. <https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads>.
- [57] York University. Hash Functions. <http://www.cse.yorku.ca/oz/hash.html>.
- [58] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2019.
- [59] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

A Case Study of Processing-in-Memory in off-the-Shelf Systems

Joel Nider¹, Craig Mustard¹, Andrada Zoltan¹, John Ramsden¹, Larry Liu¹, Jacob Grossbard¹, Mohammad Dashti¹, Romaric Jodin², Alexandre Ghiti², Jordi Chauzi², and Alexandra (Sasha) Fedorova¹

¹University of British Columbia
²UPMEM SAS

Abstract

We evaluate a new *processing-in-memory* (PIM) architecture from UPMEM that was built and deployed in an off-the-shelf server. Systems designed to perform computing *in* or *near* memory have been proposed for decades to overcome the proverbial memory wall, yet most never made it past blueprints or simulations. When the hardware is actually built and *integrated* into a fully functioning system, it *must* address realistic constraints that may be overlooked in a simulation. Evaluating a real implementation can reveal valuable insights. Our experiments on five commonly used applications highlight the main strength of this architecture: computing capability and the internal memory bandwidth *scale with memory size*. This property helps some applications defy the von-Neumann bottleneck, while for others, architectural limitations stand in the way of reaching the hardware potential. Our analysis explains why.

1 Introduction

The memory wall has plagued computer systems for decades. Also known as the *von Neumann bottleneck*, it occurs in systems where the CPU is connected to the main memory via a limited channel, constraining the bandwidth and stretching the latency of data accesses. For decades, computer scientists have pursued the ideas of *in-memory* and *near-memory* computing, aiming to bring computation closer to data. Yet most of the proposed hardware never made it past simulation or proprietary prototypes, so some questions about this technology could not be answered. When we obtained early access to soon-to-be commercially available DRAM with general-purpose processing capabilities, we recognized a unique opportunity to better understand its limitations when integrated into existing systems. Most solutions proposed in the past required specialized hardware that was incompatible in some way with currently deployed systems [2, 3, 9, 14, 16]. The hardware we evaluate was designed specifically to be used as a drop-in replacement for conventional DRAM, which has

imposed some limitations that are not present in many of the simulated architectures.

UPMEM's DRAM DIMMs include general-purpose processors, called *DRAM Processing Units* (DPU) [5]. Each 64MB slice of DRAM has a dedicated DPU, so *computing resources scale with the size of memory*. Playing to the strengths of this hardware, we ported five applications that require high memory bandwidth and whose computational needs increase with the size of the data. We observed that, indeed, application throughput scaled with data size, but scaling was not always the best that this hardware could achieve. The main reasons were the difficulty of accessing data located inside DPU-equipped DRAM from a host CPU without making a copy, the limited processing power of the DPUs, and the granularity at which the DPUs are controlled.

The main contribution of our work is understanding the limitations of PIM when it is integrated into off-the-shelf systems. We approach it via a case study of a particular implementation. We are not aware of any similar hardware that we could access, so a comparison to other architectures was outside the scope of this work. Although a comparison to other accelerators was not our goal either, we did compare with GPUs where it helped deepen our analysis.

2 Architecture

Many PIM architectures were proposed in the past [2, 3, 6–8, 10, 12, 21, 27, 28, 30, 32], ranging from logic gates embedded in each memory cell up to massive arrays of parallel processors that are located close to the memory. They all share the goals of overcoming the limitations of memory bandwidth but differ vastly in design. Nguyen et al. [22] introduce a classification system for various designs based on the distance of the compute units from the memory. We study specific hardware from the *COMPUTE-OUT-MEMORY-NEAR* (COM-N) model. This model includes compute logic that is located outside of the memory arrays but inside the memory package (same or different silicon die). This is sometimes called CIM (compute-in-memory) but is more widely known

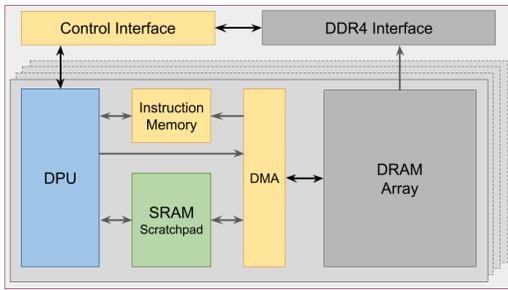


Figure 1: DPU Architecture

as PIM (processing-in-memory). After our study was concluded, Samsung announced the production of similar PIM hardware (called FIMDRAM) that also falls into the COM-N model [15]. While we could not perform an in-depth comparison, some similarities are apparent from the published design. These common design points are likely to appear in new designs as well and the lessons we learned from studying UPMEM-hardware will be generally applicable to this class of PIM.

Organization In the architecture we study, DRAM is organized into ranks (two ranks per DIMM), and each rank contains eight memory chips. Each DRAM chip includes eight DRAM Processing Units (DPUs). Each DPU is coupled with a 64MB *slice* of DRAM. There are 64 DPUs per rank and 128 DPUs per 8GB memory DIMM. Since the DPUs and DRAM are built on the same silicon die, introducing the DPUs comes at the cost of some DRAM capacity. With this design, the sum of DPUs and DRAM is limited by the silicon area on the die and the process density (number of transistors per unit area). Our experimental system has 36GB of DPU DRAM and 576 DPUs.

DPU Capabilities A DPU is a simple general-purpose processor. UPMEM’s design only supports integer operations (no floating-point hardware). Each DPU has exclusive access to its slice of DRAM over a private bus. The more total DRAM there is, the more DPUs there are. In other words, we get an additional unit of computation, and an additional bus, with each additional unit of memory – *computing capacity and intra-DIMM bandwidth scale with the size of memory*. Figure 1 shows a schematic view of a single DPU and its periphery. Before a DPU can compute on the data, it must copy it from DRAM to a 64KB private SRAM buffer, called the *working memory*. The copy is performed with an explicit, blocking DMA instruction. The DMA engine can copy up to 2KB in a single operation but the copy time increases linearly with the size. To hide DMA latency, each DPU has 24 hardware threads, called *tasklets*, that can be scheduled for execution simultaneously. Because it is an interleaved multi-threading (IMT) design [31], only one tasklet can advance at each cycle. When a tasklet is blocked on a DMA operation, other tasklets can still make progress.

DPU Speed Processing capabilities of DPUs are well below that of a common host CPU. DPUs have a simple, in-order design, and are clocked at anywhere from 266MHz (in our experimental system), to 500MHz (projected in commercial offerings). This was necessary to fit within the power envelope of commodity systems.

DPU communication In DRAM there are no communication channels between individual slices of memory dedicated to DPUs. Thus, DPUs cannot share data, other than by copying it via the host. Communication between DPUs on different chips would require additional pins in the package which would change the form factor from standard DRAM. Creating a communication bus between DPUs on the same chip is theoretically possible, but severely restricted by the number of available metal layers. Therefore, we cannot expect any PIM designs in the COM-N model to have any DPU-to-DPU communication until these problems can be solved. This constraint implies that programs running on DPUs must shard data across DPUs ahead of the execution and use algorithms that don’t require data sharing.

Interleaving At the chip level, interleaving dictates the ordering of bytes as they are written into DRAM. In the DIMMs in our test system, each DRAM chip connects to the DDR4 bus by an 8-bit interface. 8 chips together form the 64-bit wide bus. When a 64-byte cache line is committed to memory, the data is interleaved across DRAM chips at byte granularity, the first byte going into *chip 0*, the second byte into *chip 1*, etc. So when the write is done, the first chip receives bytes 0, 8, 16, ..., 56. Interleaving at this level is used to hide read and write latency by activating multiple chips simultaneously. Interleaving is transparent to the host since the data is reordered appropriately in both directions (i.e., during reads and writes). But a DPU, which can only access data in its own slice, only sees what is written to that particular chip (i.e., every *n*th byte of the data). This property makes it difficult to access the data by the host CPU and the DPU in the same location. Unless a program running on the DPU can be modified to operate on non-contiguous data, we must make a separate copy, *transposing* it in the process, so that each DPU receives a contiguous portion of the data.

Transposition UPMEM’s SDK counteracts the interleaving by transposing the data. While the cost of the transposition is small (the current SDK provides an efficient implementation that uses SIMD extensions), the cost of data copy, which requires going over the DRAM bus, is not. For applications that can place their dataset into the DPU memory and compute on it using only DPUs, the one-time cost of the copy can be negligible, but for applications where the data is short-lived or both the DPU and host CPU must access it, frequent copying will stress the memory channel – the very bottleneck this architecture aims to avoid.

Control granularity Accessing multiple chips in each bus cycle also means it is natural to schedule DPUs across those chips as a group. With the current SDK implementation, a

full rank must be scheduled as a single unit. We can only send a command to an entire rank of DPUs and cannot launch a DPU or read its memory while another DPU in the same rank is executing. Similarly, the host cannot access any memory in a rank if any DPU in that rank is executing. In practice, this results in treating DPU-equipped memory as an accelerator, rather than part of main RAM. The host typically keeps programs and data in the “normal” DRAM (i.e., *host DRAM*), copies data for processing into the *DPU DRAM*, and copies back the results. Finer-grained execution groups will likely improve the performance of algorithms that “stream” data such as *grep*.

3 Evaluation

3.1 Workloads

Considering DPU architecture properties we chose programs that need more computational resources with the increasing data size, that could be easily parallelized and where data sharing is minimal.

The SDK for the DPU architecture includes a C compiler and libraries with functions to load and launch DPUs, copy and transpose the data, etc. While we could write the code in a familiar language, the main challenges had to do with memory management (as the DPU needs to copy data from DRAM into the working memory prior to processing it), and with efficiently controlling ranks of DPUs.

Snappy Snappy is a light-weight compression algorithm designed to process data faster than other algorithms, such as *gzip*, but at the cost of a lower compression ratio [11]. Snappy divides the original file into blocks of a fixed size (typically 64KB) and compresses each block individually. The original implementation is designed for sequential (i.e., single-threaded) processing. The file format concatenates compressed blocks head to tail, without any separation marker. The result is that to decompress block n , the program must first decompress blocks $0 \dots n-1$. To enable parallelism, we changed the compressed file format by prepending each block with its compressed size. This small change enables the host program to quickly locate the start of each compressed block without having to decompress the previous blocks. We also modified the header by adding the decompressed block size. That enables us to experiment with different block sizes. Snappy is bandwidth-intensive because of the small amount of processing, especially during decompression. The majority of the time is spent copying literal strings or previously decoded strings.

Hyperdimensional computing Hyperdimensional Computing (HDC) [13] is used in artificial intelligence to model the behaviour of large numbers of neurons. It relies on vectors with high dimensionality (at least 10,000 dimensions), called *hypervectors*, to classify data and find patterns. Our HDC application performs classification of raw electromyography

(EMG) signals into specific hand gestures [25]. We built upon a prior reference implementation [20]. The classification works by computing the Hamming distance between a previously encoded input hypervector and previously trained vectors. Our implementation distributes the raw signal data among DPUs, which then encode it into hypervectors and perform the classification.

AES Encryption The Advanced Encryption Standard (AES) is a symmetric-key algorithm designed to be highly secure and run efficiently on a variety of hardware. Our analysis focuses on AES-128, using the implementation by Rijmen, Bosselaers, and Barreto found in OpenSSL [26]. Encryption is an ideal problem; the data can be divided into small individual blocks, the algorithm can operate in parallel, and the data access pattern makes it simple to optimize DMA transfers. AES can be operated in a variety of modes. We use ECB (electronic code book) mode which ensures each block can be processed individually without dependencies on the other blocks.

JSON filtering JSON is a flexible and human-readable format for storing and exchanging data. Due to its ASCII representation, parsing is notoriously slow [17, 19, 29]. When analytics systems filter records in JSON, they parse each one and check the filter condition. Parsing is performed for *all* records, even for those not meeting the filter criteria. Instead, Sparser [24] showed that it is better to filter records *before* parsing them by running string comparisons over raw (unparsed) JSON. Raw filtering is highly parallel and memory-intensive [9], and hence promising for our study. We modified Sparser [23] to offload raw filtering to DPUs, while the host performs the rest of the parsing.

Grep *Grep* [1] is a command-line utility for searching plain-text for lines that match a regular expression. We implement a subset of *grep*, which searches only for exact text matches. Our design uses the DPUs in a work pool by preparing files in small batches, and starting the search as soon as enough DPUs are ready to perform the work.

To address the challenge of DPUs needing to be controlled in large groups we must balance the work across all DPUs in the rank. By balancing the work, we minimize the difference in running time between the fastest DPU and slowest DPU, which minimizes the idle time in the rank while waiting for the slower DPUs. Preparing the work on the host consumes a significant portion of the time so it is more important to prepare the files quickly rather than efficiently packing them into DPUs. To fill the rank as evenly as possible within the time constraints, files are assigned to DPUs in a round-robin fashion in the order they appear in the input. We do not spend the time to sort the files by size, and even limit the number of files that can be processed by a single DPU to save time during preparation. A maximum of 256 files was determined empirically to have the best results.

Our baseline for performance comparison is a single host

CPU¹ since our focus is to understand when the DPU architecture overcomes the von Neumann bottleneck and when it does not.

3.2 Memory bandwidth scaling

In the DPU architecture, each additional slice of DRAM gets its own DPU along with the working memory and the DMA channel. Herein lies its main strength: *scaling computing capacity and intra-DIMM bandwidth with the size of memory*. To demonstrate, we ran a simple program where all DPUs iterate over the data in DRAM, copying every byte from their respective DRAM slices into the working memory. As shown in Fig. 2, the data processing bandwidth increases with the size of the data, reaching roughly 200GB/s for 36GB worth of DRAM (approximately 350MB/s per DPU). For a system with 128GB of DRAM and DPUs clocked at 500MHz (which is the target for commercial release) the aggregate bandwidth attained by all DPUs would reach 2TB/s.

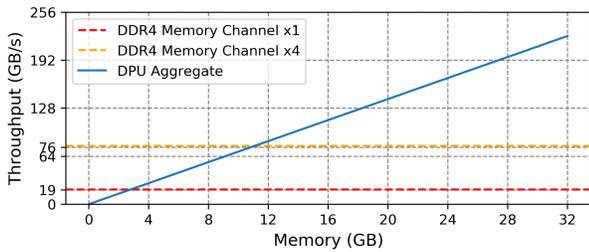


Figure 2: Memory bandwidth scales with memory capacity.

For applications whose compute and bandwidth needs grow with the size of the data, scaling of compute and bandwidth resources with memory has a direct and positive impact on performance. E.g., Fig. 3 shows the throughput of *Snappy compression* as the file size (and the number of DPUs sharing the work) increase. We use the optimal number of DPUs for each file size – i.e., using more DPUs yields no additional speedup. The input data resides in host DRAM; the experiment copies the data to DPU DRAM, allocates and launches the DPUs to run compression, and copies the compressed data back to host DRAM. As the file size grows *so does the throughput* – a direct result of compute capacity and internal bandwidth scaling with additional memory.

Breaking down the runtime, we observe that the execution time on DPUs (*Run* in Fig. 4) remains roughly the same even as we increase the file size, because more DPUs are doing the work. Again, this is the effect of scaling compute resources with data size. The DPU execution time does increase for 512MB and 1GB file sizes, because as we reach the limits of our experimental hardware, each DPU gets more data to

¹Our host machine is a commodity server with an Intel(R)Xeon(R) Silver 4110 CPU @ 2.10GHz with 64GB of conventional DDR4 DRAM.

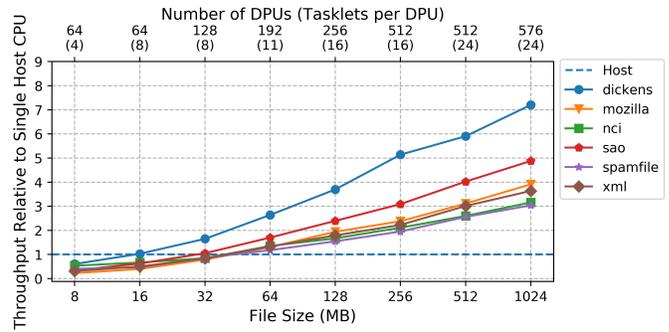


Figure 3: Snappy compression throughput

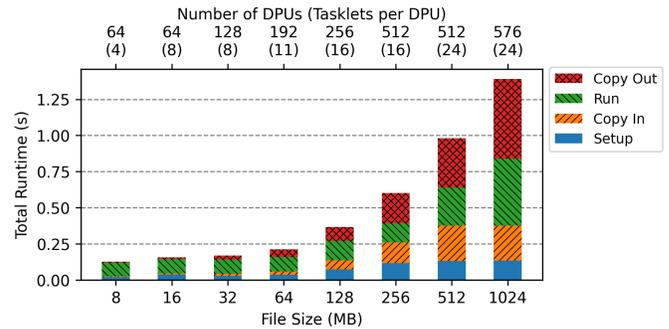


Figure 4: Runtime breakdown for compression

process. The non-scalable components of runtime in Fig. 4, *Copy In*, *Copy Out* and *Setup*, are discussed in §3.3.

A final experiment illustrating the strength of scaling involves a comparison with the GPU², an accelerator that shares similarities with DPUs (massive parallelism, high memory bandwidth), but does not have the property of scaling computational resources with memory. Fig. 5 shows the throughput relative to a host CPU (same as in the DPU experiments) of *snappy compression* as the file size grows. Contrary to the DPU architecture, the GPU scaling is flat and the throughput does not surpass that of the host CPU by more than 5×. This coincides with the internal memory bandwidth of the GPU, which is approximately 5× higher than that of the CPU.

Where each DPU has a dedicated working memory and a DMA channel, all GPU cores within an SM (streaming multiprocessor) share an L1 cache. This data-intensive workload with a random access pattern caused thrashing in the small 64KB L1 cache. Memory fetching became the bottleneck and as a result, we obtained the best performance by launching kernels consisting of a single thread per compute block (and hence one thread per SM) to reduce contention for the cache. The input file fit in the GPU memory, so there were no fetches

²We used an RTX 2070 GPU with 8GB of GDDR6 and 448GB/s global memory bandwidth. There are 36 SMs, with 64 CUDA cores per SM, clocked at 1.4GHz. There is a 64KB L1 cache per SM, and 4MB L2 cache shared by all SMs.

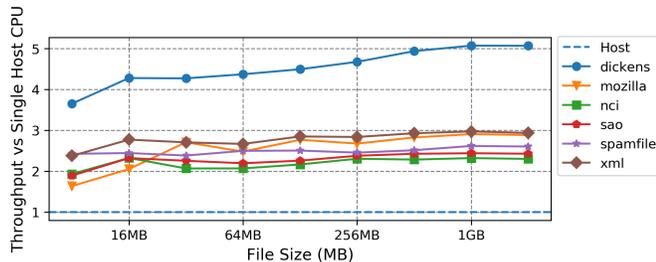


Figure 5: Snappy compression on GPUs

over the PCIe bus once processing started. In other words, the GPU fell victim to a variant of the von Neumann bottleneck. In workloads where memory bandwidth is not the main limitation, DPUs will not reach the throughput of a GPU due to their less powerful execution engine – a limitation we discuss in §3.4.

3.3 Data copy

As we discussed in §2, the need for de-interleaving (transposing) data for DPUs means that it is difficult to avoid making copies when data needs to be accessed by both DPUs and host CPUs. For applications where DPUs can be used exclusively, the data can be copied to DPU memory just once and reside there for the rest of the runtime, making copy overhead negligible. In programs where data is ephemeral or must be accessed by both the host CPU and DPUs, frequent copying will be subject to memory channel limitations – the very von Neumann bottleneck PIM aims to address.

The compression runtime breakdown presented in the previous section (Fig. 4) illustrates this point. The DPU execution time remains roughly constant as data size increases, because each DPU has a dedicated DMA channel to its DRAM slice. In contrast, the time to copy input data to DPU memory and copy the output back increases with the file size and is limited by the DRAM channel.

Setup overhead in compression does not scale, because dividing up the work requires sequential parsing of the input file. This is a property of the workload and implementation, and not fundamental to the architecture. Copy overhead, on the other hand, is fundamental to the system and is the reason why the compression throughput in Fig 3 grows sublinearly with the increasing number of DPUs.

HDC is less affected by the copy-in overhead than Snappy and has negligible copy-out cost due to the small dataset. Overall, copying takes about 10% of the total runtime, and thus HDC enjoys better scaling, as shown in Figure 6. HDC and other AI applications, where DPUs can repeatedly perform inference on long-lived data that remains in DPU memory are therefore well-positioned to circumvent the memory channel bottleneck.

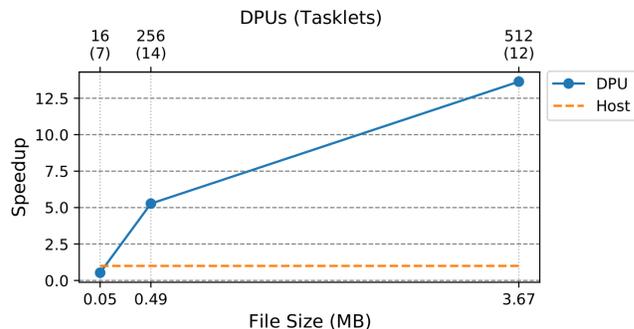


Figure 6: HDC speedup over the host as file size increases

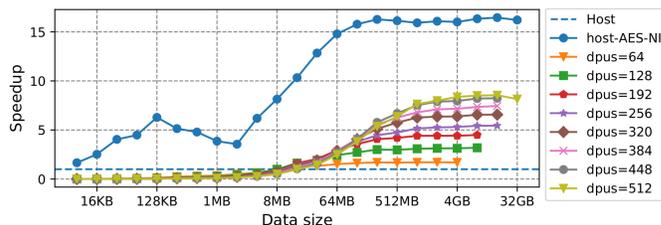


Figure 7: Encryption throughput relative to a single host CPU.

3.4 DPU Speed

Encryption provides an excellent illustration of a scenario when DPU processing power becomes a limitation. Figure 7 shows encryption throughput (bytes per second) relative to a single host CPU as the amount of data increases. We show a separate curve for each number of DPUs, to emphasize that processing power matters. Both Snappy and HDC required fewer DPUs for smaller datasets and more DPUs for larger data sets, but compute-hungry encryption relishes the highest number of DPUs we can muster, no matter the data size. Further, the far superior performance on the host with AES-NI [4] acceleration confirms that general-purpose DPUs cannot compete with specialized instruction sets.

Another example where DPU processing power was less impressive compared to other available accelerators was snappy decompression (figures omitted for brevity). On the DPU, this workload performed very similarly to compression. On the GPU, in contrast with compression, L1 cache was not the bottleneck: decompression works with smaller input block sizes and is largely write-intensive. As a result, the GPU showed similar scaling trends as on the DPU and outperformed the host CPU by up to 24×, while DPUs’ advantage was at most 8×.

3.5 Communication and control granularity

Compliance with the DDR interface and software limitations in the current SDK makes it inefficient to control DPUs one at a time: as long as a single DPU in the rank is busy, all

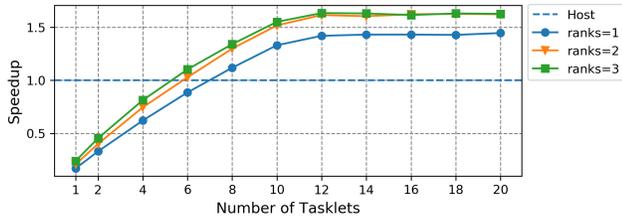


Figure 8: Speedup of `grep` over the host with varying ranks and tasklets.

others are inaccessible. This means that the program must decide how it will divide the data between DPUs in a rank and launch them all at once. While this is a trivial decision for embarrassingly parallel workloads, other workloads might under-utilize the hardware as a result. The fact that DPUs cannot communicate and steal work from one another makes matters worse. Our experience with `grep` demonstrates these challenges.

Figure 8 shows the throughput relative to a single host CPU of `grep` executed on the 875MB Linux source tree for the varying number of ranks. The throughput gain relative to the host is modest, and there is no advantage from using more than two ranks (128 DPUs). The reason is that it is very difficult to distribute the work evenly between DPUs ahead of the launch. Input source files vary in size and even organizing them into equally-sized buckets per DPU does not guarantee that the processing time on all DPUs will remain similar. As a result, `grep` used the hardware inefficiently: the ratio of execution times on the fastest and slowest DPUs was 116 \times , indicating the presence of stragglers that held up the entire rank. In contrast, `JSON filtering`, a workload very similar to `grep` but where dividing the work evenly was easy, enjoyed the fastest/slowest execution ratio of 1 (no stragglers) and similar scaling properties as the rest of the workloads (figure omitted for brevity).

3.6 System cost

We conclude with the analysis of system cost, showing that for applications with high bandwidth requirements, a system that uses DPU memory is cheaper than the one without it.

With a memory-bound workload, increasing the number of processors without increasing the memory bandwidth does not improve performance by a significant amount [3, 18, 33]; we need a CPU with a high number of memory channels that can supply the cores with data and avoid starvation. CPUs are designed with a wide range of cores but do not include additional memory channels for a higher number of cores. Therefore, we must use more sockets and faster memory to scale up the memory bandwidth in a CPU-only system. Table 1 shows an estimated price comparison of system configurations with the maximum possible memory bandwidth with

off-the-shelf Intel systems (1.6TB/s) assuming Snappy compression as the representative workload. We use the nominal price of \$60 for an 8GB DIMM and \$300 for an 8GB PIM module [5]. The costs shown include CPU, memory and PIM (where applicable). We assume DDR4-2400 (19.2 GB/s) with a Xeon 4110 (\$500) for all configurations except the last, which uses DDR-3200 (25.6 GB/s – 33% more bandwidth) and a Xeon 8380 (\$8100). With our conservative estimations, a CPU-only system is less expensive for applications below 500GB/s. However when scaling up, a CPU-only configuration is approximately 3.8 \times more expensive to attain the same Snappy compression throughput.

DRAM	PIM	Snappy throughput	Cost	Cost per MB/s
Low bandwidth requirements				
48GB	-	512MB/s	\$860	\$1.68
48GB	32GB	512MB/s	\$2160	\$4.21
High bandwidth requirements				
48GB	448GB	7168MB/s	\$17660	\$2.46
448GB	-	7263MB/s	\$68640	\$9.45

Table 1: System cost per MB/s for Snappy compression.

4 Conclusion

Memory-bound applications are limited by the conventional memory hierarchy, and adding more CPUs does not improve performance substantially. We have shown several real-world applications that can be accelerated by PIM by scaling memory bandwidth with compute resources. PIM shares many attributes (such as massive parallelism) with other accelerators but the distribution of processing units inside the memory gives it a unique advantage in certain cases. The architecture we evaluated has some limitations that prevent exploiting the full capabilities of the hardware, which remain as challenges for future designs. Costly data transfers which must be performed in order to share data between the host CPU and PIM memory adds significant overhead to processing time but can be mitigated by reusing data in-place. Despite the limitations, we find that PIM can be effective when applied to the correct class of problems by scaling the processing power with the size of the dataset.

Acknowledgements

Thank you to our shepherd, Heiner Litz, for his helpful comments and suggestions to improve this paper. A special thank you to Vincent Palatin for his patience and support explaining the details of this new hardware.

Availability

All source code for the projects described in the paper can be found at <https://github.com/UBC-ECE-Sasha>. The UPMEM SDK is publicly available at <https://www.upmem.com/developer/>.

References

- [1] Tony Abou-Assaleh and Wei Ai. Survey of global regular expression print (grep) tools. In *Technical Report*, 2004.
- [2] Sandeep R Agrawal, Sam Idicula, Arun Raghavan, Evangelos Vlachos, Venkatraman Govindaraju, Venkatanathan Varadarajan, Cagri Balkesen, Georgios Giannikis, Charlie Roth, Nipun Agarwal, and et al. A many-core architecture for in-memory data processing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 245–258, New York, NY, USA, 2017. Association for Computing Machinery.
- [3] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyong Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 105–117, New York, NY, USA, 2015. Association for Computing Machinery.
- [4] Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, and Ronen Zohar. Breakthrough AES performance with Intel® AES new instructions. In *Intel Whitepaper*. Intel, 2010.
- [5] F. Devaux. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–24, 2019.
- [6] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. The architecture of the diva processing-in-memory chip. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, page 14–25, New York, NY, USA, 2002. Association for Computing Machinery.
- [7] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. Computedram: In-memory compute using off-the-shelf dram. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 100–113, New York, NY, USA, 2019. Association for Computing Machinery.
- [8] M. Gao and C. Kozyrakis. Hrl: Efficient and flexible reconfigurable logic for near-data processing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 126–137, March 2016.
- [9] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development*, 63(6):3:1–3:19, 2019.
- [10] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: the terasys massively parallel pim array. *Computer*, 28(4):23–31, April 1995.
- [11] Google. Snappy - a fast compressor/decompressor. *Goole Inc.*, 2020.
- [12] Roman Kaplan, Leonid Yavits, and Ran Ginosar. From processing-in-memory to processing-in-storage. In *Supercomput. Front. Innov. : Int. J.* 4, 2017.
- [13] Geethan Karunaratne, Manuel Le Gallo, Giovanni Cherubini, Luca Benini, Abbas Rahimi, and Abu Sebastian. In-memory hyperdimensional computing. In *Nature Electronics*, 2020.
- [14] Liu Ke, Udit Gupta, Carole-Jean Wu, Benjamin Youngjae Cho, Mark Hempstead, Brandon Reagen, Xuan Zhang, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, and Xiaodong Wang. Recnmp: Accelerating personalized recommendation with near-memory processing, 2019.
- [15] Young-Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, O Seongil, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, Youngmin Cho, Jin Guk Kim, Jongyoon Choi, Hyun-Sung Shin, Jin Kim, BengSeng Phuah, HyoungMin Kim, Myeong Jun Song, Ahn Choi, Daeho Kim, SooYoung Kim, Eun-Bong Kim, David Wang, Shinhaeng Kang, Yuhwan Ro, Seungwoo Seo, JoonHo Song, Jaeyoun Youn, Kyomin Sohn, and Nam Sung Kim. 25.4 a 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2tflops programmable computing unit using bank-level parallelism, for machine learning applications. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 350–352, 2021.
- [16] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. Tensor-dimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 740–753, New York, NY, USA, 2019. Association for Computing Machinery.

- [17] Geoff Langdale and Daniel Lemire. Parsing gigabytes of json per second. *The VLDB Journal*, 28(6):941–960, 2019.
- [18] Dominique Lavenier, Charles Deltel, David Furodet, and Jean-François Roy. BLAST on UPMEM. Research Report RR-8878, INRIA Rennes - Bretagne Atlantique, March 2016.
- [19] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. Mison: A fast json parser for data analytics. *Proc. VLDB Endow.*, 10(10):1118–1129, June 2017.
- [20] F. Montagna, A. Rahimi, S. Benatti, D. Rossi, and L. Benini. Pulp-hd: Accelerating brain-inspired high-dimensional computing on a parallel ultra-low power platform. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.
- [21] Amir Morad, Leonid Yavits, Shahar Kvatinsky, and Ran Ginosar. Resistive gp-simd processing-in-memory. *ACM Trans. Archit. Code Optim.*, 12(4), January 2016.
- [22] Hoang Anh Du Nguyen, Jintao Yu, Muath Abu Lebdeh, Mottaqiallah Taouil, Said Hamdioui, and Francky Catthoor. A classification of memory-centric computing. *J. Emerg. Technol. Comput. Syst.*, 16(2), January 2020.
- [23] Shoumik Palkar, Firas Abuzaid, and Justin Azoff. Sparser Source Code.
- [24] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Filter before you parse: Faster analytics on raw data with sparser. *Proceedings of the VLDB Endowment*, 11(11), 2018.
- [25] A. Rahimi, S. Benatti, P. Kanerva, L. Benini, and J. M. Rabaey. Hyperdimensional biosignal processing: A case study for emg-based hand gesture recognition. In *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–8, 2016.
- [26] Vincent Rijmen, Antoon Bosselaers, and Paulo Barreto. Optimised ansi c code for the rijndael cipher (now aes), 2000. https://github.com/openssl/openssl/blob/master/crypto/aes/aes_core.c.
- [27] Patrick Siegl, Rainer Buchty, and Mladen Berekovic. Data-centric computing frontiers: A survey on processing-in-memory. In *Proceedings of the Second International Symposium on Memory Systems, MEMSYS '16*, page 295–308, New York, NY, USA, 2016. Association for Computing Machinery.
- [28] H. S. Stone. A logic-in-memory computer. *IEEE Transactions on Computers*, C-19(1):73–78, Jan 1970.
- [29] Tencent. RapidJSON. <https://rapidjson.org/>.
- [30] K. Wang, K. Angstadt, C. Bo, N. Brunelle, E. Sadredini, T. Tracy, J. Wadden, M. Stan, and K. Skadron. An overview of micron’s automata processor. In *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–3, Oct 2016.
- [31] W. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *The 16th Annual International Symposium on Computer Architecture*, pages 273–280, May 1989.
- [32] Dongping Zhang, Nuwan Jayasena, Alexander Lyshevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. Top-pim: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC '14*, page 85–98, New York, NY, USA, 2014. Association for Computing Machinery.
- [33] Keira Zhou, Jack Wadden, Jeffrey J. Fox, Ke Wang, Donald E. Brown, and Kevin Skadron. Regular expression acceleration on the micron automata processor: Brill tagging as a case study. In *Proceedings of the 2015 IEEE International Conference on Big Data (Big Data), BIG DATA '15*, page 355–360, USA, 2015. IEEE Computer Society.

XFUSE: An Infrastructure for Running Filesystem Services in User Space

*Qianbo Huai**, *Windsor Hsu**, *Jiwei Lu**, *Hao Liang[†]*, *Haobo Xu** and *Wei Chen**

**Alibaba Group
Sunnyvale, California
USA*

*[†]Alibaba Group
Shenzhen, Guangdong
China*

Abstract

Implementing the filesystem in user space reduces development complexity [28, 30] and decreases dependency on the underlying OS platform. Implementing the filesystem at the user level as opposed to inside the OS kernel, however, has traditionally meant lower performance [11, 17, 22]. This performance overhead is increasingly limiting with high performance storage devices based on new persistent memory technology (e.g. 3D XPoint [13]) and advanced networking techniques (e.g. RDMA [14]). User space file systems have also been associated with poor reliability, availability and serviceability (RAS) characteristics [26]. As a result, there is a tendency to consider user space filesystems as prototypes and proof-of-concepts. In this paper, we systematically analyze the concerns with deploying user space filesystem to provide production file storage services. We present XFUSE, a filesystem in user space framework that addresses the performance and RAS concerns, and that enables file storage services to be effectively deployed at the user level. Our performance analysis indicates that XFUSE enables filesystem requests made through standard kernel interfaces to be processed at the user level with latency in the 4 microseconds range, and offers throughput exceeding 8 GB/s.

1 Introduction

User space code is generally easier to develop and maintain than kernel code. Thus filesystems with advanced functionality tend to be developed as user space filesystems (e.g. [3, 5, 24, 31, 32]). However, because filesystems are traditionally incorporated into the OS kernel, applications have been largely developed using standard kernel filesystem interfaces. This means that user space filesystems will incur additional performance overhead from passing messages between the kernel and the user space filesystem [11, 17, 22]. This overhead is especially limiting with the use of high performance storage devices and networking that routinely offer latency in the microseconds range and throughput of several GB/s [13, 14].

The use of user level networking [16, 18] and I/O helps to reduce the overall impact of crossing into user space to reach the filesystem but the performance hit is still significant. In particular, metadata operations such as stat and other operations that benefit from filesystem caching may be fast in kernel mode filesystems, but will be slower in user space filesystems due to additional communication cost between the operating system kernel and user space file systems [22].

Moreover, ensuring reliability, availability and serviceability (RAS) for user space filesystems has additional complexity because the rest of the system may continue to operate when the filesystem is down. For example, if a user space filesystem aborts, the application using the filesystem may continue to execute and wait for its requests to be completed. This partial failure possibility requires additional handling but it also provides a basis for the user space filesystem to be upgraded without disrupting the application. Such non-disruptive upgrade facilitates the deployment of new releases in production environments. There has been some previous work [26] on supporting restartable user space filesystems. However, the prior work requires significant kernel changes and do not support advanced filesystem features such as direct I/O and multi-threading.

Workloads are increasingly executed on the cloud in virtual machines (VMs) and sandboxed containers [4, 8, 12] to enable efficient resource utilization and increase agility. Cloud service providers can offer additional storage services to their customers through a storage client such as a filesystem gateway to object storage, optimized NAS client, etc. For the aforementioned reasons, it is advantageous to implement this storage client in user space. Furthermore, if the storage client can be deployed outside of the customer VMs and in the VM host, cloud service providers will be able to clearly separate the storage client from the user VM and independently manage the client as part of the storage service. There has been some recent work in this area. For example, virtio-fs [9] builds upon FUSE to allow VMs to access a host filesystem directory.

In this paper, we focus on enabling the deployment of pro-

duction storage services using user space filesystems. We evaluate prior approaches to understand what remains to be done to make this a reality. We propose XFUSE, a software framework patterned after FUSE [7] that addresses the performance and RAS concerns generally associated with user space filesystems. We believe that the improvements that XFUSE provides over FUSE can be applied to FUSE-based approaches such as virtio-fs [9] to better support running the user space filesystem in the VM host as opposed to inside the VMs that are running user applications.

Our contributions include:

- Systematically analyze the path a kernel filesystem request takes from being issued by an application to being handled by a user space filesystem and have the results communicated back to the application.
- Design and implement an optimized framework for user space filesystem that is backward compatible with FUSE and that takes advantage of the growing number of cores available on modern systems to achieve low latency and high throughput for fast storage devices.
- Demonstrate that such a framework enables kernel filesystem requests to be processed in user space with latency in the 4 microseconds range and throughput exceeding 8 GB/s.
- Extend the framework to provide features such as support for online upgrade and crash recovery that are critical for deploying user level filesystems in production.

The rest of this paper is organized as follows. In the next section, we survey related work. In Section 3, we introduce the design and implementation of XFUSE. In Section 4, we evaluate XFUSE performance. Section 5 concludes this paper.

2 Background and Related Work

FUSE is a widely adopted framework to support a filesystem in user space [7]. Figure 1 shows its highlevel architecture. FUSE consists of a Linux kernel module and a user space library that is linked to a user space filesystem daemon process. Conceptually, the FUSE kernel module consists of two parts: filesystem handler which is the code that processes filesystem operations including the code that invokes VFS, and device handler which interacts with the user space filesystem via a special device, `/dev/fuse`.

The filesystem handler and device handler run in different process contexts. The former runs in the context of the application process while the latter runs in the filesystem daemon process context. Communication between the two parts is coordinated through kernel events and incurs context switch cost. More specifically, there is a context switch when an application process sends a filesystem request to the filesystem daemon process, and there is another context switch when

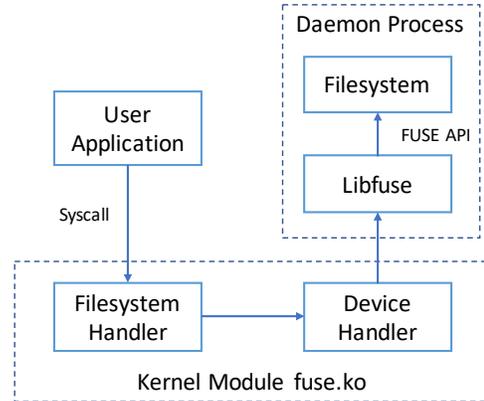


Figure 1: FUSE (and XFUSE) Architecture.

the filesystem daemon process responds to the application’s request. Therefore each filesystem operation submitted by the application incurs at least two context switches with FUSE. In addition, FUSE has a single pending request queue from which threads of a filesystem daemon process pick up incoming requests for processing. Under load, this global queue becomes a source of contention.

There has been a large body of work on analyzing and improving the performance of FUSE (e.g. [11, 17, 22, 30]). ExtFUSE [11], for instance, extends FUSE so that a filesystem in user space can register a piece of simple code in the OS kernel to handle selected filesystem operations without incurring a context switch. There have also been alternatives to FUSE. For example, AVFS [1] uses the environment variable `LD_PRELOAD` [15, 21] to intercept libc POSIX API entry and invoke filesystem operations without context switch.

More recently, ZUFS [10] has been proposed as an alternate framework for user space filesystem that eliminates one data copy by having the kernel module copy data directly from the source to the destination. ZUFS and XFUSE share many of the same performance goals. We attempted to evaluate ZUFS but encountered issues. Our queries, as well as those from others, on the ZUFS project page on GitHub went unanswered. It appears that ZUFS is no longer maintained.

There has also been work on implementing the filesystem as an embeddable library running in user space (e.g. NVFUSE [6]) and on providing the filesystem as a separate user space process that communicates with each application through a private communication channel (FSP [19]). These approaches require applications to be rebuilt. They also bypass the VFS layer which provides important functionality such as as permission checking, file sharing coordination and buffer management.

The idea of transparently restarting the filesystem upon failure is explored in [25] and a specific framework that provides support for restartable user space filesystems is proposed in [26]. This framework, however, requires significant kernel

changes and does not support advanced filesystem features such as direct I/O and multi-threading [26]. The shadow driver concept proposed in [27] inspired us to keep track of incoming requests and eventually led to the crash restart algorithm proposed in this paper.

There has been a lot of recent interest in enabling VMs (and sandboxed containers [4, 8, 12]) to share a host directory. In particular, virtio-fs [9] builds upon FUSE to allow a VM to access files on the host. It provides a direct access (DAX) mode whereby data in the host page cache can be mapped into a VM and then accessed directly from within the VM. For non-DAX mode operations, virtio-fs is essentially FUSE with virtio [23] as the channel between the filesystem and device handlers. From this perspective, we believe that the improvements XFUSE provides over FUSE should be applicable to virtio-fs as well.

3 XFUSE

XFUSE is designed to achieve low latency and high throughput for fast storage devices, and scale with the increasingly large number of CPU cores available in today’s systems. Besides supporting user space filesystems in general, XFUSE is specifically designed for deploying the following types of filesystems:

- Filesystems that use high speed storage devices such as those based on persistent memory technology. To effectively leverage the very low read and write latency (microseconds range) that these devices offer, software overhead must not dominate.
- Filesystems that use SSDs and distributed storage systems based on high performance network technology such as RDMA. Such storage systems can deliver low I/O latency (on the order of 100 microseconds) and high throughput (several GB/s).
- Filesystems that are used in a production environment where availability of service is critical and disruption of users should be kept to a minimum. In particular, recoverable faults and maintenance activities such as filesystem upgrade should not materially impact service to the user.

FUSE has been widely adopted to deploy user space filesystem. Thus XFUSE is designed to be backward compatible with FUSE. To end users, an XFUSE mount is almost identical to a FUSE mount. To filesystem developers, XFUSE supports the FUSE API and extends it to enable additional functionality such as support for online upgrade and crash restart. Just as with FUSE, XFUSE consists of a kernel module, `xfuse.ko`, and a library, `libxfuse.a`, that needs to be linked into the filesystem daemon.

3.1 Performance

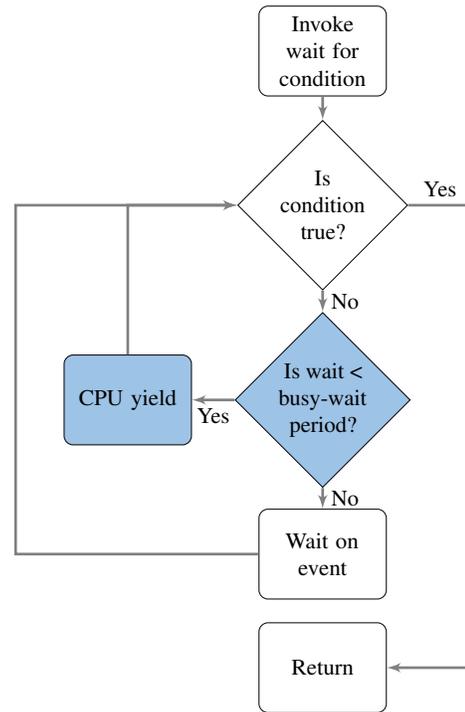


Figure 2: Busy-Event Wait.

3.1.1 Adaptive Waiting

XFUSE is patterned after FUSE [7] and has a similar architecture. Figure 1 can be used to describe the XFUSE architecture as well. From the figure, a FUSE request flows from the user application to the filesystem daemon process through the filesystem handler and the device handler. After the filesystem has processed the request, it sends a response in the reverse direction through the device handler and the filesystem handler. FUSE uses kernel events to coordinate the flow between the filesystem and device handlers. The latency for a FUSE request thus includes the time required by the filesystem daemon to handle the request and the time for two event waits - one by the device handler to obtain an incoming request and the second by the request handler to obtain the response from the filesystem daemon.

A kernel event notification takes on the order of a few microseconds to be delivered. For requests that can be handled quickly by the filesystem, the overhead of two event waits is very costly. For example, the latency of filesystem metadata requests that operate on in-memory data is dominated by the two event waits. The event waits also mask the low-latency benefit provided by high performance storage devices such as those based on persistent memory technology.

To avoid this overhead, XFSUSE introduces an initial period of busy waiting or busy polling to the event wait. This wait scheme is depicted in Figure 2. If the condition being sought by a process is satisfied during the busy waiting period, the process continues without incurring the cost of an event wait. However, if the condition being sought is not met within the busy-wait period, the scheme falls back to regular event wait. We term this scheme *busy-event wait*.

With XFSUSE, the device handler running in a filesystem daemon thread checks for new requests in a busy loop for a short period of time. If a pending request arrives during this busy waiting period, it is found and handled immediately by the daemon thread. If no new request arrives within the short period of time, the thread falls back to waiting on an event that is signalled whenever a request is submitted. The device handler sends a reply back to the filesystem handler in the same manner. If the filesystem daemon can handle a request fast enough, the filesystem handler may still be busy waiting when the reply becomes available. In this case, the end-to-end latency as observed by the application process can be as low as 3-4 microseconds.

The effectiveness of busy waiting has some dependence on whether the application threads and filesystem daemon threads are running on the same or different CPUs. We evaluate this factor in Section 4.1.2. In setting the busy-wait period, there is a tradeoff between request latency and CPU utilization which affects throughput. The busy-wait period should be set based on the performance characteristics of the filesystem and the underlying storage system. Later in Section 4.1.1, we evaluate this parameter and find that a busy-wait period on the order of 10 microseconds achieves a good balance for systems based on persistent memory technology as well as those based on SSD.

To further reduce CPU consumption and potentially increase throughput, we can dynamically adjust the busy-wait period based on the latency observed in the system. In particular, if the actual time required to service a request exceeds the busy-wait period, attempting to busy wait is futile and only wastes CPU resources. We thus experimented with a simple scheme that dynamically disables busy waiting whenever the last observed latency exceeds the busy-wait period by more than the net overhead of event wait, and reenables it whenever the last observed latency falls below this threshold. We refer to this scheme as *adaptive busy-event wait*. Results reported in Section 4.1.1 show that adaptive busy-event wait is very effective at avoiding unnecessary busy waiting, thereby increasing throughput.

3.1.2 Increased Parallelism

With the growing number of cores available on modern systems, increasing the number of requests that can be processed in parallel is the key to increasing throughput. XFSUSE enables multiple filesystem daemon threads to work on different

requests in parallel, and supports the asynchronous processing model to enable each thread to handle multiple concurrent requests.

More specifically, XFSUSE provides multiple communication channels between the filesystem handler and device handler. Each channel has two queues. One is the free queue containing request slots that can be used for new requests. The other is the processing queue containing requests that are inflight. The filesystem handler selects a channel for an incoming request using a channel selection policy. At the other end of each channel is a filesystem daemon thread waiting for new requests. There can be multiple requests in the processing queue per daemon thread. XFSUSE also ensures that threads working on different channels do not have lock contention between them. The number of channels and the number of request slots per channel determine the maximum concurrency that XFSUSE can deliver to the filesystem daemon. In order to make effective use of the multiple channels and associated filesystem daemon threads that XFSUSE supports, thread placement and channel selection policies are important considerations. We discuss and evaluate these policies later in Sections 4.1.2 and 4.1.3.

In contrast, FUSE uses a single request queue to hold incoming requests. When there are many concurrent accesses, the single request queue may become a source of contention and limit the throughput that FUSE can achieve. As we shall see later in Section 4, XFSUSE is able to drive much higher throughput than FUSE through increased parallelism.

3.2 RAS

3.2.1 Online Upgrade

Scheduling service downtime to perform an upgrade is very disruptive in production settings. By operating outside of the kernel, a user space filesystem can potentially be upgraded without disrupting the application. Such a non-disruptive or online upgrade capability will ease the introduction of new features and big fixes, and facilitate the deployment of user space filesystems in production environments.

When a filesystem daemon terminates, all the file descriptors to the special device `/dev/fuse` are closed. This leads the kernel to unmount the filesystem automatically. In order to keep the kernel from unmounting the filesystem during an upgrade of the filesystem daemon, XFSUSE provides a monitor service to hold all the XFSUSE device file descriptors while the old filesystem daemon exits and a new daemon executing the upgraded software takes over to serve running applications.

XFSUSE includes a library, `libxfuse` to facilitate the interaction between the filesystem daemon and the XFSUSE monitor service. `Libxfuse` extends the FUSE `libfuse` library with new functionalities and APIs that allow user space filesystems built with it to support online upgrade.

Figure 3 illustrates the XFSUSE-assisted filesystem online

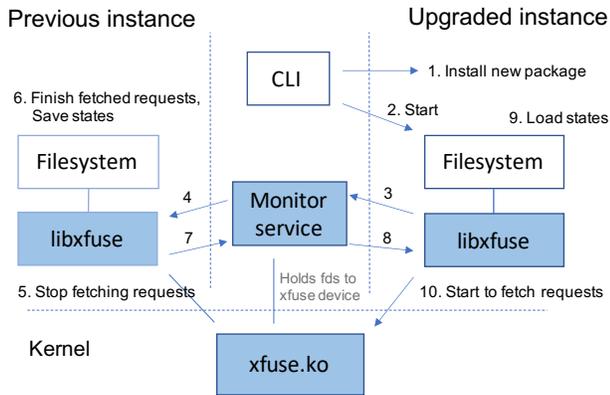


Figure 3: Filesystem Online Upgrade Workflow.

upgrade workflow. Figure 4 shows the XFUSE state transitions in both filesystem daemons during the upgrade process. The numbered actions in the two figures match. Certain details, such as error handling, are omitted to reduce clutter.

1. The upgrade workflow starts with an operation to update the filesystem software package. In Figure 3, this operation is triggered through a CLI (command line interface) command to install the new software package.
2. After installation, a filesystem daemon process running the new software is started. This filesystem daemon initializes its XFUSE stack with parameters to interact with the XFUSE monitor service, and provides libxfuse with a number of callback functions. The usage of those callbacks is depicted in Figure 4.
3. Upon starting, libxfuse connects to the XFUSE monitor service on behalf of the filesystem daemon and fetches all the file handles to the XFUSE device. As discussed earlier, having the monitor service create and hold XFUSE device handles ensures that the filesystem service can remain online to applications during the upgrade.
4. When a filesystem daemon is running, it maintains a communication channel with the XFUSE monitor service. Through this channel, the monitor service notifies the current filesystem daemon that it is being upgraded.
5. Libxfuse in the current filesystem daemon stops fetching new requests from the kernel. However, XFUSE device read operations issued moments earlier may still be returning from the kernel. The filesystem may also be replying to just-processed requests. Libxfuse waits for all pending requests to drain.
6. The filesystem continues processing requests. When libxfuse is certain that there are no more inflight requests, it notifies the filesystem to save its transient state. As an

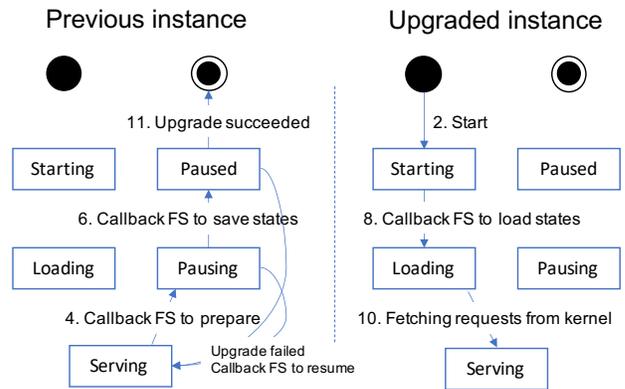


Figure 4: State Transitions during Filesystem Online Upgrade.

optimization, libxfuse can notify the filesystem earlier in Step 4 to start preparing for an upgrade so as to reduce the time needed to save the transient state after incoming requests are paused.

7. Libxfuse notifies the XFUSE monitor service on behalf of the current filesystem daemon that it is paused and that its transient state has been saved. The saved state includes open file handles, file locks and all other runtime data necessary for a new filesystem daemon to take over and continue serving running applications.
8. The XFUSE monitor service notifies the new filesystem daemon that the saved state is ready to be loaded.
9. Libxfuse in the new filesystem daemon calls back into the filesystem to load the saved state and prepare to handle incoming requests.
10. The new filesystem daemon starts to fetch requests from the kernel. Past this point, the previous filesystem daemon exits. The online upgrade has been successfully completed.

We have implemented a user space filesystem with the ability to save and restore its transient state, and integrated it with XFUSE to enable us to upgrade the filesystem software with minimal user impact. The filesystem and XFUSE components are deployed and serving production workloads. In Figure 5, we plot the performance as observed by the fio benchmark [2] operating against the filesystem as the filesystem is upgraded. At 34 seconds into the plot, the upgrade is initiated. This triggers a series of steps to verify and install the new software package. A filesystem daemon process running the new software is then started. Throughout this time, the current filesystem daemon continues to serve I/O requests. At 55 seconds into the plot, the current filesystem daemon is asked to stop processing incoming requests. By 57 seconds into the plot, the current filesystem daemon has completed all

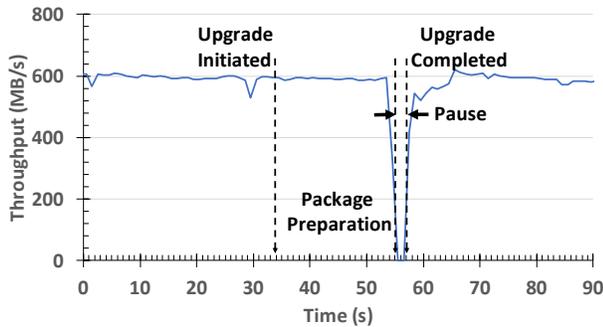


Figure 5: Throughput During Online Upgrade of Filesystem.

pending requests and saved all of its transient state. The new filesystem daemon loads the saved transient state and starts to serve incoming requests. At this point, the old filesystem has served its purpose and terminates itself. There is a ramp up in performance as the new filesystem establishes its cache.

3.2.2 Crash Restart

The online upgrade process forms the basis for supporting crash restart of the filesystem daemon process. Just as for online upgrade, in order to support crash restart, the filesystem must be able to remember its transient state and reinstate it via the new filesystem daemon instance. After the new instance is up and has restored the previous state, it can start to replay any pending requests. The kernel module does not know the exact point at which the filesystem daemon process crashed and restarted. It simply resends all the pending requests that it has already sent to the previous instance and for which it has not received the response.

The new filesystem daemon instance, however, cannot simply re-process all the requests because filesystem requests are not idempotent. For example, suppose a user successfully removes a file *P*. If the file removal operation were to be applied again, it would fail because the file *P* no longer exists.

To handle crash restart in the face of such non-idempotent requests, the filesystem daemon saves information about recent requests in a request-response table. This table records the response for requests that have already been processed by the filesystem daemon. For each incoming request, the filesystem daemon replies with the saved response if the request is recorded in the table. Otherwise it processes the requests normally. To uniquely identify a request, the kernel module assigns a request ID, which encodes a consecutive sequence ID, to each incoming request. To ensure that IDs are assigned consecutively, the filesystem daemon needs to persist the largest request ID it has processed. After a restart, the new daemon instance passes this value as a mount flag to the kernel.

The size of the request-response table is bounded by the number of requests that may be in flight. Conceptually, the XFUSE kernel module keeps a queue of the requests that

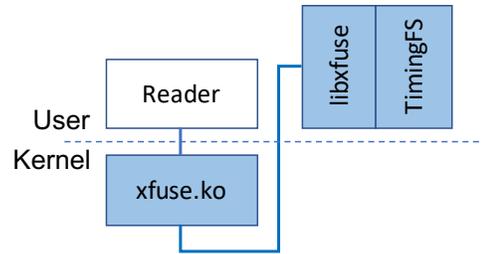


Figure 6: Experimental Setup for Parametric Analysis.

it has received the response for. When sending the next request, it dequeues a completed request and sends its request ID along with the new request. The filesystem daemon uses the completed request ID to prune its request-response table. In practice, the tracking of completed requests and their responses is done on a per channel basis to avoid introducing points of contention in this process. Each channel only needs to keep track of up to queue depth number of recent requests.

4 Performance Evaluation

To evaluate the performance characteristics of XFUSE, we first use a controlled environment to explore the different aspects of XFUSE individually. The goal of this analysis is to systematically understand how these aspects are affected by policy choices and tuning parameters, and to project the performance that can potentially be achieved by a filesystem daemon that is optimized for XFUSE. We then measure actual system performance on real systems when the filesystem requests are looped through XFUSE and FUSE, and compare the results with directly accessing kernel-mode EXT4 [20].

4.1 Parametric Analysis

In this section, we use the experimental setup depicted in Figure 6. The setup is designed to provide a controlled environment where various aspects of XFUSE can be isolated and the associated parameters can be tuned systematically. Because existing user space filesystems have not been optimized beyond what FUSE can drive, this setup also serves to project the kind of performance that XFUSE can potentially achieve with a user space filesystem that is optimized for it. The setup consists of a reader composed of multiple reader threads each synchronously reading 4 KB of data from a random location in a large file. These read requests are sent via XFUSE to TimingFS, a simple filesystem daemon that emulates the timing characteristics of persistent memory and SSD.

TimingFS supports only `readdir`, `getattr` and `read` operations. `Readdir` and `getattr` operations are supported only to the extent necessary for a filesystem to be mounted and a file within the filesystem to be opened for read. The main purpose of TimingFS is to respond to read requests of the opened file. In

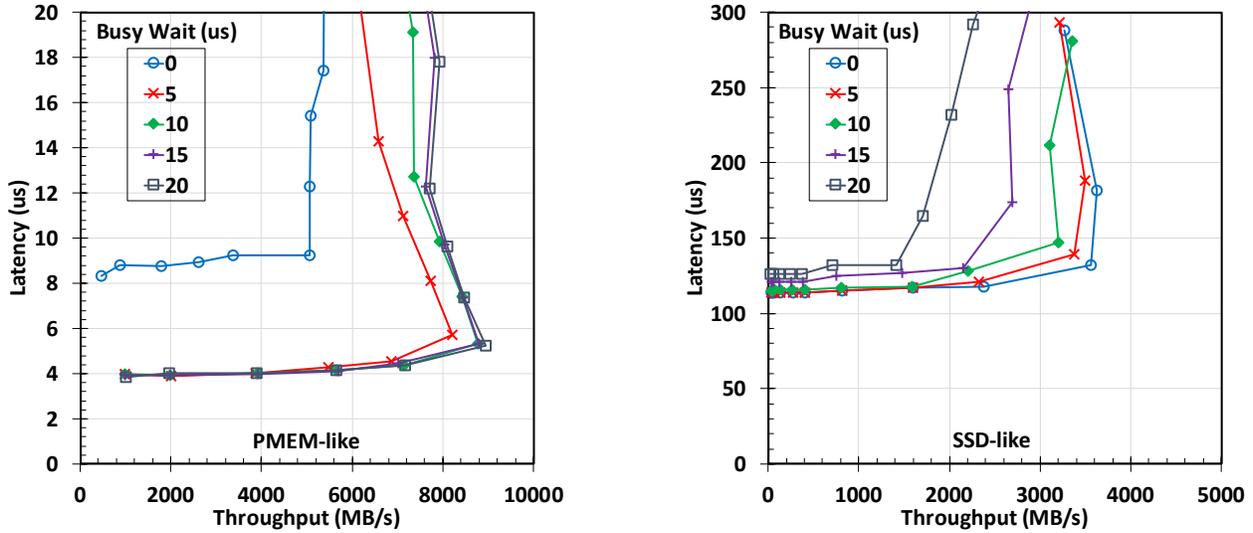


Figure 7: Performance with Busy-Event Wait.

the persistent memory mode, TimingFS handles the file read requests by immediately copying the requested amount of data from the specified location in a memory pool standing in as the file. In the results, this case is denoted as PMEM-like. In the SSD mode, TimingFS enqueues the read request and a worker thread handles the enqueued request after a delay of 100us. This case is denoted as SSD-like in the results. We use a file size of 1 GB to ensure that the memory footprint exceeds the L1 and L2 processor caches as is typically the case for I/O requests because of the amount of data involved. For 4 KB requests, ensuring that the data does not reside solely in the processor caches adds about 1 us to the read latency.

All the experiments in this analysis were performed using dedicated servers running Linux 4.19.91 on the Alibaba Cloud. Each server has dual Intel(R) Xeon(R) Platinum 8163 CPUs operating at 2.50GHz for a total of 48 physical cores. We restricted the experiments in this section to using the first 24 physical cores.

4.1.1 Waiting Strategy

Figure 7 summarizes the effect of introducing an initial period of busy waiting to event wait. Note that with a busy-wait period of 0, busy-event wait degenerates into event wait. Observe that adding on the order of 10 us of busy waiting is very effective at lowering latency and increasing throughput for PMEM-like storage. On the other hand, adding busy waiting is not effective for SSD-like storage. In fact it degrades performance significantly for SSD-like storage, and the degradation increases with the busy-wait period. This is because the service time with SSD-like storage exceeds the busy-wait period so that the busy waiting is wasted and only serves to drive up CPU utilization which affected the throughput.

We next investigate enabling busy waiting only when it

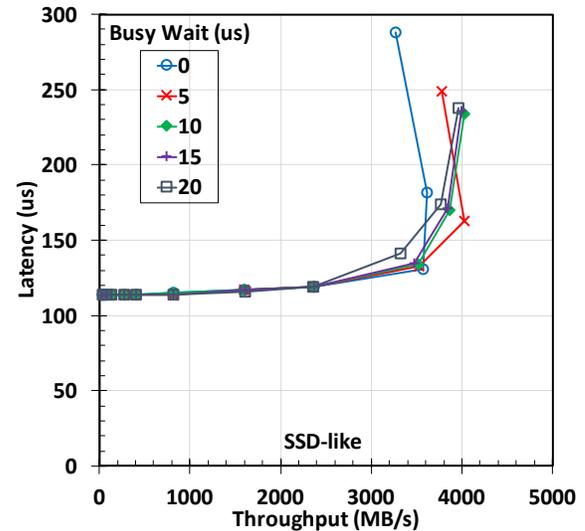


Figure 8: Performance with Adaptive Busy-Event Wait.

is beneficial. The basic idea is that if the current wait is going to be longer than the busy-wait period, we should skip the busy waiting and go straight to event waiting. We use a simple method of predicting the current wait time based on the observed latency of the last completed wait. Now, the last wait could have been completed by an event notification so a conservative threshold for enabling busy waiting is that the last observed latency is within the busy-wait period plus the net overhead of event wait. In other words, we disable busy waiting when the last observed latency exceeds the busy-wait period by more than the net overhead of event wait and reenables it when the last observed latency falls below this threshold. From Figure 9, the latency achieved with busy

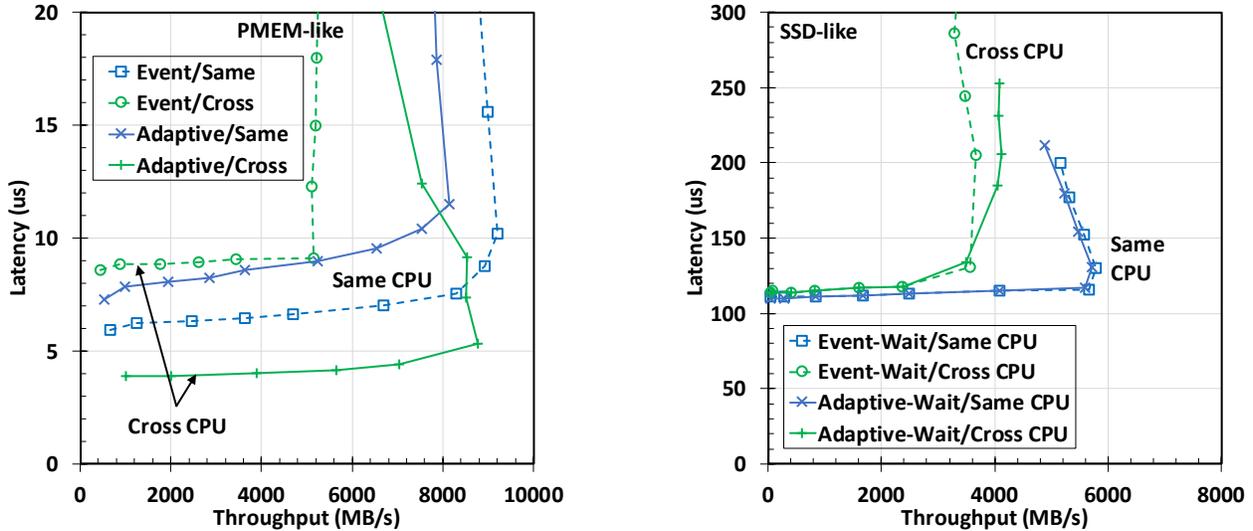


Figure 9: Effect of Thread Placement.

waiting can reach just under 4 us while that with event waiting is around 9 us, suggesting that the net event overhead is about 5 us in our experimental setup.

As shown in Figure 8, with such an adaptive busy-event wait scheme, the latency becomes relatively insensitive to the busy-wait period because futile busy waiting is avoided. With adaptive busy-event wait, a busy-wait period in the 10 us range works well for both the PMEM-like and SSD-like configurations. Observe further that throughput is increased beyond event-wait even though the service time with SSD-like storage far exceeds the busy-poll periods under consideration. Recall that there are two waits in the system. One is by the device handler to obtain an incoming request and the second is by the request handler to obtain a response from the filesystem daemon process. For SSD-like storage, the device handler running in the filesystem daemon thread to pick up incoming requests benefits from the busy waiting when under load. Thus adaptive busy-event wait offers a performance benefit even for relatively slow SSD-like storage.

4.1.2 Thread Placement

The overhead of busy waiting and event waiting depends on whether the application threads and filesystem daemon threads are executing on the same CPU. If the application thread and corresponding filesystem daemon thread run on different CPUs and use busy waiting to send messages to each other, the number of context switches will be greatly reduced. On the other hand, if the two threads are scheduled on the same CPU, context switches between the 2 threads are needed to accept the request and receive the response. Furthermore, the scheduler typically implements a per CPU run queue so that local CPU event notification is likely to be delivered faster. There are also cache locality considerations when it

comes to scheduling the application threads and filesystem daemon threads, but our evaluation shows that this effect is secondary.

In Figure 9, we investigate the effect of thread placement on performance by controlling the CPU on which each application and filesystem daemon thread runs, and the mapping of requests to channels. Specifically, we affine each application thread to a CPU and map each request to a channel based on the CPU ID of the application thread issuing the request. On the filesystem daemon side, we consider two cases - one where the thread listening on a channel is affined to the same CPU as the application thread whose requests are mapped to that channel (denoted same CPU), and the second where the listening thread is affined to a different CPU (denoted cross CPU). Observe that for PMEM-like storage, cross CPU busy waiting offers the lowest latency of just under 4 us, outperforming same CPU busy waiting by between 3-5 us. On the other hand, same CPU event waiting significantly outperforms cross CPU event waiting. In the SSD-like case, the long service time with SSD-like storage means that busy waiting does not improve latency. Thus for SSD-like storage, the lowest latency is obtained by scheduling the application threads and corresponding filesystem daemon threads on the same CPU.

In some production environments such as those where large server farms are used to provide a specific set of services to many customers, the thread placement on each server can be controlled as we have done in these experiments. When PMEM-like storage is used in these types of environments, the application and corresponding filesystem daemon threads should be placed on different CPUs and adaptive busy-event wait should be used so as to achieve a significant improvement in latency. For SSD-like storage, the application and corresponding filesystem daemon threads should be placed

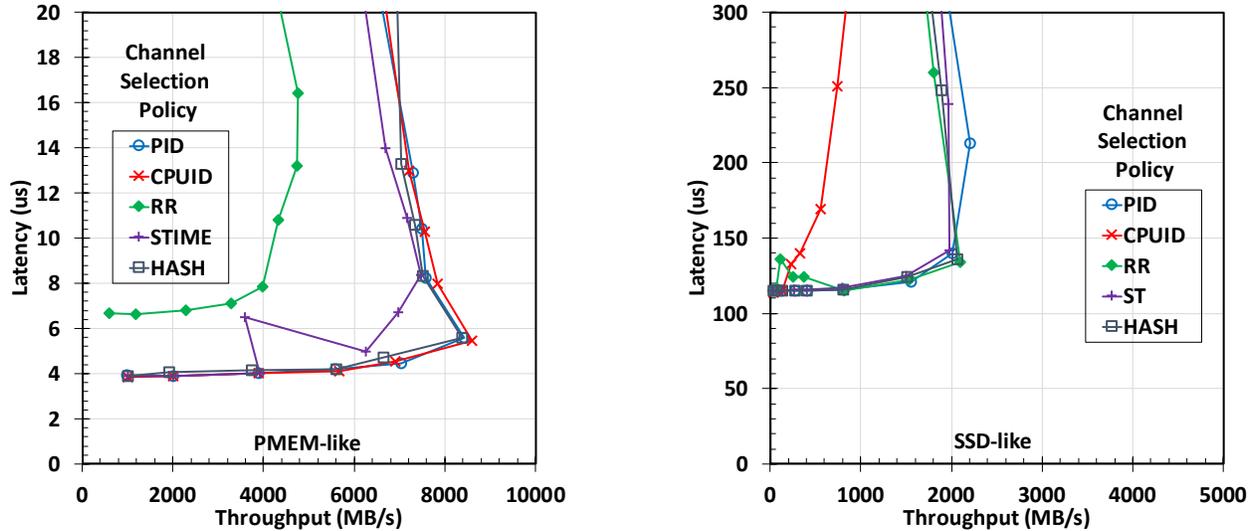


Figure 10: Effect of Channel Selection Policy.

on the same CPU. In this case, adaptive busy-event wait does not improve performance but does not hurt either.

4.1.3 Channel Selection

In other environments where thread placement cannot be explicitly managed, the performance achieved depends on how the scheduler maps the application and filesystem daemon threads to CPUs, and on how the requests are mapped to the channels. In Figure 10, we evaluate various channel selection policies when the thread placement is left entirely up to the scheduler.

The best strategy in this case is to evenly distribute requests across the channels so that multiple filesystem daemon threads can be brought to bear on handling the requests. There is one caveat in that if the policy keeps switching to an idle channel, it will render busy waiting ineffective. In the figure, RR denotes the round robin channel selection policy. In the PMEM-like case where busy waiting can be effective, RR performs worse than other policies because it keeps rotating to a channel where the corresponding filesystem daemon thread is no longer busy waiting. In order to leverage busy waiting, the channel selection policy needs to have a bias towards a specific channel.

We also evaluate using the CPU ID, thread ID and thread start time to map requests to channels. Depending on where the scheduler places the application threads, using the CPU ID to select channels may result in a very skewed channel distribution. Thread start time may also collide and result in a less than balanced distribution. We find that using thread ID is much more stable in terms of yielding an even distribution because the thread ID is at least unique. To further avoid the risk of a skewed distribution, we can use hashing techniques on the thread ID to avoid colliding on the same channel. The

HASH selection algorithm uses 3 hash functions to identify candidate channels and selects the channel with the shortest queue. In the case of a tie, it always selects the first candidate channel to avoid defeating busy waiting. HASH consistently avoids skewed distribution.

4.1.4 Performance Potential

XFUSE is designed to provide an efficient conduit between the kernel file system interface and a user space filesystem daemon. The performance that it can deliver ultimately depends on the capability of the user space filesystem. Here, we use our simple filesystem daemon, TimingFS, to project the best-case performance that XFUSE can achieve with a user space filesystem that is optimized for it. We configure XFUSE based on the analysis results discussed above. Specifically, we use adaptive busy-event wait with a busy-wait period of 10 us and an event overhead of 5 us. We also set up XFUSE with 24 channels and corresponding TimingFS threads, one for each of the physical cores that are used for the experiments.

Figure 11 presents the results. Notice that in the PMEM-like configuration, XFUSE is able to achieve latency in the 4 us range and throughput exceeding 8 GB/s. FUSE, on the other hand, has a latency of 10 us and throughput of less than 2 GB/s. Across both PMEM-like and SSD-like configurations, XFUSE significantly outperforms FUSE both in terms of latency and throughput. The improvement is especially dramatic for the PMEM-like configuration because the storage is so fast that even small overheads become hugely significant.

4.2 System-Level Performance

In this section, we use system-level benchmark workloads to evaluate the performance potential of XFUSE. The experi-

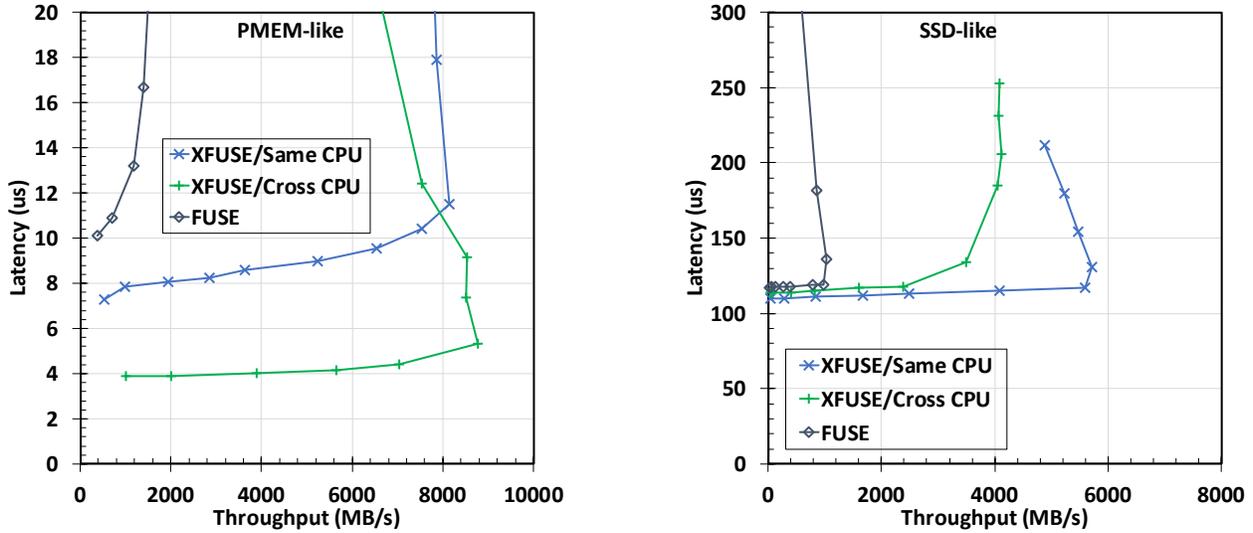


Figure 11: Performance of XFUSE and FUSE with TimingFS.

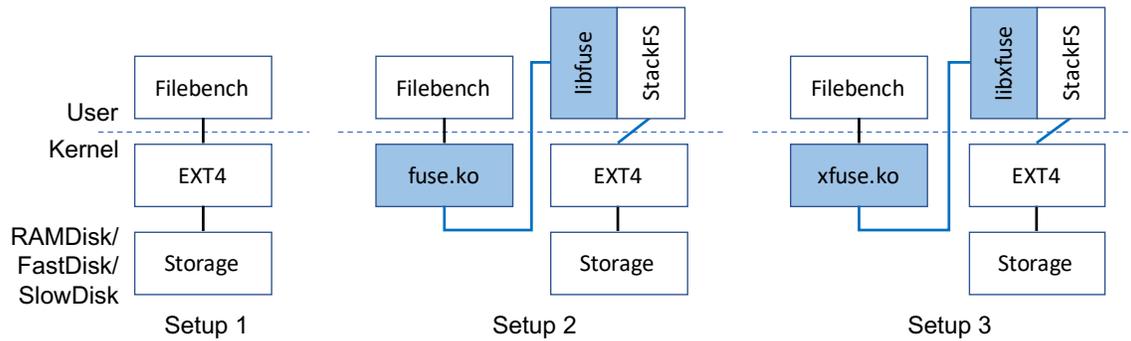


Figure 12: Experimental Setup for Evaluating System-Level Performance.

mental setup is depicted in Figure 12. The setup is designed foremost to provide a common basis for comparing XFUSE with FUSE and regular kernel-mode EXT4, and errs on the side of being conservative for XFUSE. Specifically, in this setup, XFUSE and FUSE rely on StackFS [30] to route calls back to kernel-mode EXT4. StackFS is a simple stackable passthrough filesystem originally developed to evaluate FUSE performance. StackFS has not been reworked to take advantage of the low-latency and high parallelism that XFUSE can deliver.

As discussed in [30], existing techniques such as the kernel page cache and FUSE’s readahead are effective at masking the performance of user space filesystems in several types of workload. In this section, we present results focusing on those cases where FUSE has a significant gap with kernel-mode EXT4, namely the Filebench random read, file create and webserver workloads, as previously defined and made public [29, 30]. We ran these workloads as is and without specifying CPU affinity.

We performed the measurements using dedicated servers running Linux 4.19.91 on the Alibaba Cloud. The FUSE version used is 3.6.1. Each server has 48 physical cores (dual Intel(R) Xeon(R) Platinum 8163 CPUs operating at 2.50GHz) and 768GB of memory. All the experiments in this section were limited to using the first 24 physical cores and 256 GB of memory. The remaining 512 GB of memory was allocated as a RAM disk. We configured XFUSE based on the analysis results presented in the previous section, meaning that we used adaptive busy-event wait with a busy-wait period of 10 us and an event overhead of 5 us. We also set up XFUSE with 24 channels and corresponding StackFS threads, one for each of the physical cores.

The first config uses the RAM disk as the storage device. This config is meant to illustrate the potential performance achievable when using XFUSE to access a user mode persistent memory based filesystem. We refer to this setup as RAMDisk. In the second config, we use the fastest cloud disk available. The average latency for a 4 KB read is about 115 us

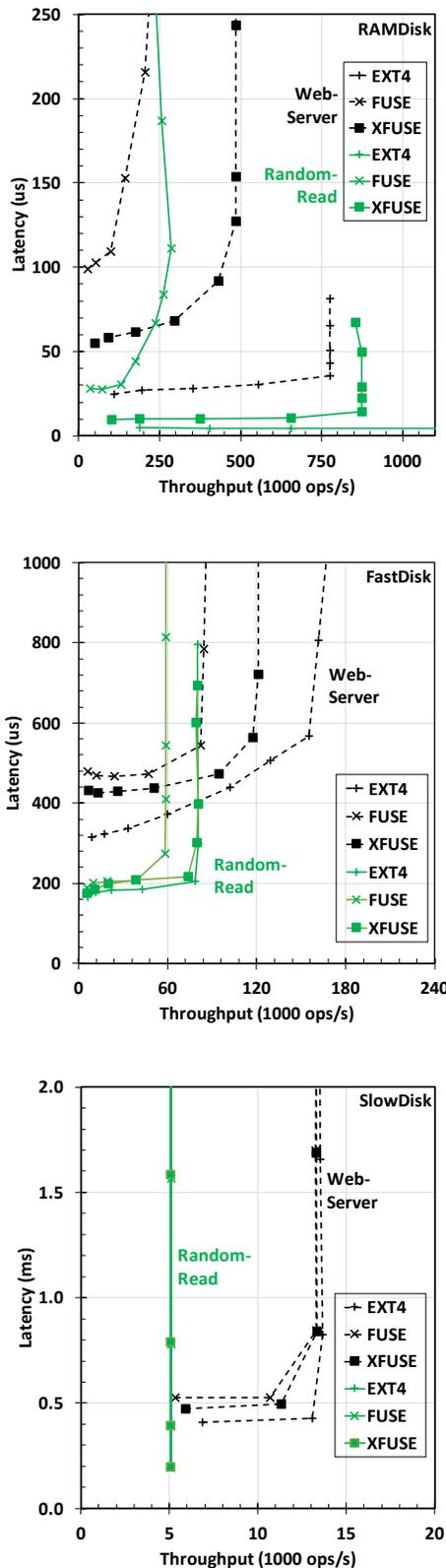


Figure 13: Performance of Random-Read and Web-Server Workloads.

on this disk. Based on its provisioned size, the throughput for this disk is capped at 80K IOPS. We refer to this setup as FastDisk. In the third config, we use the slowest cloud disk available. The average 4 KB read latency is about 250 us for this disk and it is limited to 5K IOPS. We refer to this setup as SlowDisk.

Figure 13 summarizes the performance results for the Random-Read and Web-Server workloads. With RAMDisk, kernel-mode EXT4 outperforms FUSE by a large margin across the 2 workloads. XFUSE closes the gap significantly both in terms of latency and throughput. For Random-Read, latency with XFUSE comes in at 9 us versus 28 us with FUSE. Throughput with XFUSE exceeds 874K ops/s while FUSE can only achieve 285K ops/s. This represents a 3x improvement. The improvement is similarly large for the Web-Server workload. The gap with kernel-mode EXT4 is still significant as expected because the XFUSE results are obtained by looping requests to user space and back into kernel-mode EXT4. We expect the gap to be reduced with an actual filesystem daemon running in user space that is optimized for XFUSE.

Observe that as the storage is slower, the gap between kernel-mode EXT4 and FUSE shrinks as does the benefit of XFUSE over FUSE. With SlowDisk, there is virtually no performance difference between XFUSE and FUSE. This is to be expected because when the storage is slower, the performance is bottlenecked more by the storage than by the conduit to user space. For FastDisk, which is the storage that performance-critical workloads are most likely to be using, XFUSE offers significant benefit over FUSE. For example, for the Web-Server workload, latency with XFUSE is 425 us versus 466 us with FUSE, and throughput is 125K ops/s versus 86K ops/s with FUSE. Note that the throughput of FastDisk is capped at 80K IOPs based on its provisioned size. For the Random-Read workload, XFUSE is able to deliver the full throughput of FastDisk, matching the throughput achieved by kernel-mode EXT4.

The results for the File-Create workload are summarized in Figure 14. To reduce clutter, the results for FastDisk are not presented. XFUSE outperforms FUSE for both FastDisk and RAMDisk, but not by as large a margin as in the case of the Random-Read and Web-Server workloads. For the File-Create workload, the throughput with kernel-mode EXT4 is several times higher than that with either FUSE or XFUSE. The File-Create workload creates millions of small files and each create requires several calls into StackFS to perform getattr and pathname lookups. StackFS in turn dynamically allocates memory and uses internal global locks and states to translate the requests it receives back to POSIX calls to the underlying EXT4. These roundtrips between the kernel and StackFS as well as the implementation of StackFS restrict the performance of File-Create with FUSE and XFUSE. Notice further from the figure that the File-Create workload does not scale in performance beyond 4 threads on kernel-mode EXT4. This limits the benefit that XFUSE can provide over

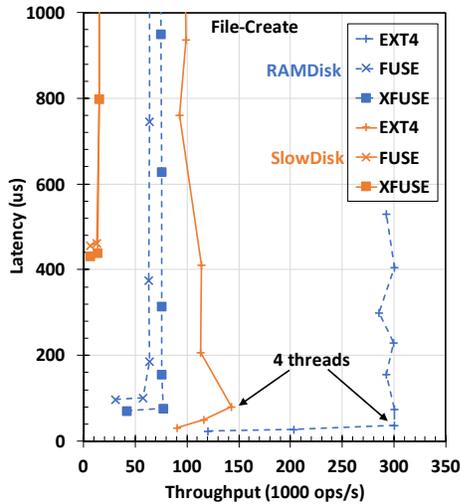


Figure 14: Performance of File-Crete Workload.

FUSE because the increased parallelism that XFUSE provides cannot be brought to bear on this workload with EXT4 as the backing filesystem.

The results underscore the point that XFUSE is designed to provide an efficient conduit between the kernel filesystem interface and a user space filesystem. The performance that it can achieve depends ultimately on the capability of the user space filesystem. Our measurements indicate that even with an existing FUSE-targeted StackFS, XFUSE offers significant performance improvement over FUSE. We anticipate that the improvement will be even more pronounced with user space filesystems that are designed to take advantage of the low latency and high parallelism that XFUSE can deliver.

5 Conclusion

This paper presents XFUSE, a user space filesystem framework that addresses the performance and RAS concerns generally associated with user space filesystems. XFUSE is backward compatible with FUSE and takes advantage of the growing number of cores available on modern systems to achieve low latency and high throughput for fast storage devices. XFUSE can enable filesystem requests made through standard kernel interfaces to be processed at the user level with latency in the 4 microseconds range, and offers throughput exceeding 8 GB/s. XFUSE also provide features such as support for on-line upgrade and crash recovery that are critical for deploying user level filesystems in production.

Acknowledgments

This paper would not have been possible without the collaboration of our storage and kernel teams. We are grateful to

our shepherd and anonymous reviewers for helping us improve the paper, and to the many who have worked on FUSE and on whose shoulders we stand. We are working towards contributing XFUSE to the community.

References

- [1] AVFS - A Virtual File System. <http://avf.sourceforge.net>.
- [2] fio - Flexible I/O tester rev. 3.27. https://fio.readthedocs.io/en/latest/fio_doc.html.
- [3] FUSE-based file system backed by Amazon S3. <https://github.com/s3fs-fuse/s3fs-fuse>.
- [4] gVisor is an application kernel for containers that provides efficient defense-in-depth anywhere. <https://gvisor.dev>.
- [5] A network filesystem client to connect to SSH servers. <https://github.com/libfuse/sshfs>.
- [6] NVMe based File System in User-space. <https://github.com/nvfuse/nvfuse>.
- [7] The reference implementation of the Linux FUSE (Filesystem in Userspace) interface. <https://github.com/libfuse/libfuse>.
- [8] Secure and fast microVMs for serverless computing. <https://firecracker-microvm.github.io>.
- [9] virtio-fs. <https://virtio-fs.gitlab.io>.
- [10] zero-copy file-system feeder. A Linux module which dispatch kernel's VFS commands to user-space server. <https://github.com/NetApp/zufs-zuf>.
- [11] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference*, pages 121–134, 2019.
- [12] Jay Chen. *Making containers more isolated: An overview of sandboxed container technologies*, (accessed January 12, 2020). <http://unit42.paloaltonetworks.com/making-containers-more-isolated-an-overview-of-sandboxed-container-technologies>.
- [13] Intel Corp. *Intel® Optane™ technology for data centers*, (accessed January 12, 2020). <http://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/optane-for-data-centers.html>.

- [14] 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.
- [15] Shi Zhan Feng Dan Zhao Heng and Yao Yingying. LD_PRELOAD based file system management. *Journal of Huazhong University of Science and Technology (Natural Science Edition)*, 2010.
- [16] Yukai Huang, Jinkun Geng, Du Lin, Bin Wang, Junfeng Li, Ruilin Ling, and Dan Li. LOS: A high performance and compatible user-level network operating system. In *Proceedings of the First Asia-Pacific Workshop on Networking*, pages 50–56, 2017.
- [17] Shun Ishiguro, Jun Murakami, Yoshihiro Oyama, and Osamu Tatebe. Optimizing local file accesses for FUSE-based distributed storage. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 760–765, 2012.
- [18] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: A highly scalable user-level tcp stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation*, pages 489–502, 2014.
- [19] Jing Liu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Sudarsun Kannan. File systems as processes. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.
- [20] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: Current status and future plans. In *Proceedings of the Linux Symposium*, volume 2, pages 21–33, 2007.
- [21] Kevin Pulo. Fun with LD_PRELOAD. In *linux.conf.au*, volume 153, 2009.
- [22] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 206–213, 2010.
- [23] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.
- [24] Frank B Schmuck and Roger L Haskin. GPFS: A shared-disk file system for large computing clusters. In *USENIX Conference on File and Storage Technologies (FAST’02)*, 2002.
- [25] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Michael M Swift. Membrane: Operating system support for restartable file systems. *ACM Transactions on Storage (TOS)*, 6(3):11, 2010.
- [26] Swaminathan Sundararaman, Laxman Visampalli, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Refuse to crash with Re-FUSE. In *Proceedings of the sixth Conference on Computer Systems*, pages 77–90, 2011.
- [27] Michael M Swift, Muthukaruppan Annamalai, Brian N Bershad, and Henry M Levy. Recovering device drivers. *ACM Transactions on Computer Systems (TOCS)*, 24(4):333–360, 2006.
- [28] Vasily Tarasov, Abhishek Gupta, Kumar Sourav, Sagar Trehan, and Erez Zadok. Terra incognita: On the practicality of user-space file systems. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, 2015.
- [29] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.
- [30] Bharath Vangoor, Kumar Reddy, 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, 2017.
- [31] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 307–320, 2006.
- [32] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Fast*, volume 8, pages 1–14, 2008.

MAX: A Multicore-Accelerated File System for Flash Storage

Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu*

*Department of Computer Science and Technology, Tsinghua University
Beijing National Research Center for Information Science and Technology (BNRist)*

Abstract

The bandwidth of flash storage has been surging in recent years. Employing multicores to fully unleash its abundant bandwidth becomes a necessary step towards building high performance storage systems. This paper presents the design and implementation of MAX, a multicore-friendly log-structured file system (LFS) for flash storage. With three main techniques, MAX systematically improves the scalability of LFS while retaining the flash-friendly design. First, we propose a new reader-writer semaphore to scale the user I/Os with negligible impact on the internal operations of LFS. Second, we introduce file cell to scale the access to in-memory index and cache while delivering concurrency- and flash-friendly on-disk layout. Third, to fully exploit the flash parallelism, we advance the single log design with runtime-independent log partitions, and delay the ordering and consistency guarantees to crash recovery. We implement MAX based on the F2FS in the Linux kernel. Evaluations show that MAX significantly improves scalability, and achieves an order of magnitude higher throughput than existing Linux file systems.

1 Introduction

The bandwidth of solid-state drives (SSDs) has been quickly increasing over the past decade [23, 24, 55]. To unleash full throughput potentials from such improvement, efficiently utilizing multicores to handle concurrent I/Os becomes a must. Currently, Non-Volatile Memory Express (NVMe) protocol [9] and multi-queue block layer [13] have already laid a multicore-friendly foundation at the driver layer. Additionally, in the upper software stack, great efforts have been made to increase the scalability [29, 38, 40, 43, 50, 56].

Nonetheless, an important question is still left unanswered: whether the log-structured file systems (LFS) atop the flash-based SSDs adapt well to the scaling of cores. LFS, initially introduced in Sprite LFS [52], builds on a simple idea: organizing the address space as an append-only log. This design essentially converts random writes into sequential ones, which not only aligns with the I/O preference of legacy hard

disk drive (HDD), but also is a common practice of file system for flash storage [35, 37, 49]. First, due to the intrinsic NAND idiosyncrasies, the sequential performance of most flash SSDs is still significantly higher than the random performance [24–28, 53, 54]. Second, the zoned namespace (ZNS), an optimized interface for flash SSDs, is available in NVMe spec and under increasing promotion [4, 6, 12, 58]. ZNS favors log-structured writes, and existing LFSes (e.g., F2FS [37]) can directly run atop it. Therefore, LFS is a promising architecture for flash SSD, and understanding its performance in the multicore context yields great significance.

Hence, we start this paper with a study on file systems (esp. LFS) throughput under concurrent and independent I/Os (§2). By increasing the number of CPU cores, we observe that most file systems scale relatively well on traditional storage devices (i.e., HDD and SATA SSDs). Surprisingly, the performance of file systems atop the modern NVMe SSDs suffers greatly from scaling. Most notably in F2FS, an LFS optimized for NVMe SSD, the performance peaks at only 18 cores, and a further scaling to 72 cores causes throughput drop by nearly 30%. The I/O utilization of F2FS on NVMe SSD is only 20%.

Through profiling, we conclude that the unscalable data structures of the file system cost a considerable amount of CPU cycles and thus bottleneck the performance. The root cause of the inefficiency comes from a legacy choice: using shared data structures to aggregate file operations and I/Os for high performance. Such philosophy essentially trades CPU cycles for high device I/O utilization. However, for NVMe SSD, this trade-off breaks as the CPU cycles are no longer negligible for high performance drives.

While many research have been conducted to understand and improve such inefficiency [29, 48, 56], they mostly focus on journaling file systems (e.g., Ext4 [44]) and therefore can not directly be applied to LFS due to different designs. For instance, LFS uses a checkpoint mechanism instead of journaling for persistence and consistency, thereby unable to use techniques such as parallel journaling for scalability [29, 56]. Hence, we first analyze the root causes inside the LFS.

Here, we decompose the LFS internals from top to bottom into three levels, the Concurrency Control (CC) level, the In-Memory Data Structure (IMDS) level and Space Allocation (SA) level, as shown in Table 1. We find that the lock con-

*Jiwu Shu (shujw@tsinghua.edu.cn) is the corresponding author.

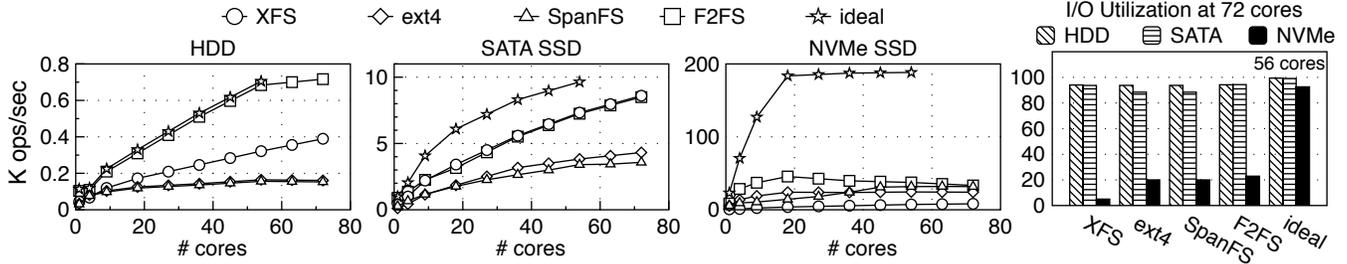


Figure 1: **Scalability problem evaluation.** We make two major observations: (1) On slow hard disk drive and SATA SSD, most file systems scale well, and the storage device is almost saturated. (2) In contrast, on high performance NVMe SSD, the performance of almost all file systems starts to drop after 18 cores, and the device is underutilized. HDD: Seagate ST1000NX0313; SATA SSD: Samsung 850 Pro; NVMe SSD: Intel DC P3700. Described in §2.

Lock level	Shared mode	Exclusive mode
Concurrency Control (CC)	write operations (e.g., write)	global operations (e.g., sync)
In-Memory Data Structure (IMDS)	index read operations (e.g., write)	index write operations (e.g., create)
Lock level	Mutual exclusion lock	
space allocation (SA)	durability operations (e.g., fsync)	

Table 1: **Current practices of the file system sharing.**

tentions caused by the shared data structures in each level serialize independent I/Os, and further prevent applications from taking full advantage of multicore-friendly design of NVMe and the abundant bandwidth of flash storage.

In this paper, we introduce the following three design principles to reduce the lock contentions of each level, and further scale the multicore performance of LFS for flash storage (§3).

- In CC level, parallelizing the external I/O requests (e.g., read and write syscalls) while keeping the internal operations (e.g., LFS checkpoint) efficient.
- In IMDS level, scaling the IMDS access while delivering flash- and concurrency-friendly on-disk format for concurrent persistence functions (e.g., fsync).
- In SA level, paralleling persistence functions while delegating (some) ordering and consistency to crash recovery.

We implement these ideas in MAX (§4), a multicore accelerated file system for high performance flash storage, with a set of modifications of F2FS. MAX replaces a global reader-writer semaphore (rwsem) of F2FS with a tailored rwsem (§4.1), reorganizes the IMDS (§4.2) and on-disk layout (§4.3) and slightly changes the crash recovery procedure (§4.4). During the development of MAX, we surprisingly find some optimizations of F2FS restrict the multicore performance. We describe our solutions to maintain these optimizations while offering better scalability (§4.5). In a nutshell, MAX restructures F2FS, enabling independent I/Os to concurrently enter the file system, concurrently access the IMDS and concurrently reach the persistent storage. We further study MAX’s performance (§5). Under a wide variety of micro- and macro-benchmarks, we show that MAX achieves up to an order of magnitude higher

throughput than existing Linux file systems. Further, evaluation against memory-backed tmpfs [51] indicates that, for certain file operations (e.g., appending blocks to private files), the performance of MAX almost reaches the upper bound of VFS (virtual file system).

In summary, we provide three major contributions.

- We perform a study on the multicore scalability of LFS and demonstrate that the lock contentions from different levels as the major culprits.
- We propose MAX removes unnecessary sharing of different levels by using a novel reader-writer semaphore, a new in-memory data structure abstraction and log partitions.
- Our evaluations show that MAX outperforms existing Linux file systems by up to an order magnitude. For some file operations, MAX comes close to the upper bound of VFS.

2 Background and Motivation

2.1 Understanding the Performance

We start with measuring the throughput of 5 file system setups (XFS [18], Ext4 [44], SpanFS [29], F2FS [37] and ideal) on three types of storage devices (HDD, SATA SSD and NVMe SSD). In the ideal setup, we partition the drive and run an independent F2FS on each partition. This enables each parallel process to execute in its dedicated file system without software level contention. As fdisk [2] only allows at most 56 partitions, we do not include performances of ideal setup beyond that. §5 further describes other details of the testbed.

In the experiment, we attach each core with one process and scale the number of cores from 1 to 72 (i.e., X axis in Figure 1). Each process runs for 60 seconds, and executes the following operations. First, the process creates a file in its own directory. Then the process issues 4 KB writes, invokes the fsync on each file and then deletes the file.

Figure 1 plots the results. First, we observe that existing file systems scale poorly on NVMe SSDs. For HDD and slower SATA SSD, the performance of many file systems (e.g., F2FS and XFS) is close to that of the ideal setup. For NVMe SSDs, the throughput of existing file systems is far from ideal. Instead of benefiting from the scalability, most file

Operations	Lock/Sharing	Overhead	Lock level	Description
<code>write()</code>	<code>sbi->cp_rwsem</code>	50.98%	CC	Mutual exclusion between checkpoint and write operations.
<code>create()</code>	<code>nm_i->nat_tree_lock</code>	3.74%	IMDS	Protecting a radix tree indexing of inode table.
	<code>sbi->inode_lock</code>	3.22%	IMDS	Protecting a list indexing of dirty inode.
	<code>curseg->curseg_mutex</code>	1.84%	SA	Serializing log-structured space allocation.
	<code>nm_i->nid_list_lock</code>	1.09%	IMDS	Protecting a central ever-increasing inode ID allocator.
<code>unlink()</code>	<code>nm_i->nat_tree_lock</code>	22.68%	IMDS	Protecting a radix tree indexing of inode table.
	<code>im->ino_lock</code>	6.54%	IMDS	Protecting a list and radix tree indexing of cached inode.
	<code>sbi->inode_lock</code>	3.05%	IMDS	Protecting a list indexing of dirty inode.
	<code>node_inode</code>	2.66%	IMDS	A user-invisible inode structure to trace all inode pages.
	<code>sit_i->sentry_lock</code>	2.41%	SA	Synchronizing concurrent access to segment info table.
<code>fsync()</code>	<code>sbi->writepages</code>	45.76%	SA	Enforcing the sequential log access.
	<code>sit_i->sentry_lock</code>	1.07%	SA	Synchronizing concurrent access to segment info table.

Table 2: **The scalability bottlenecks of F2FS.** `write()`: overwriting blocks of private files, `create()/unlink()`: creating/deleting files in private directories, `fsync()`: invoking `fsync` on private files. Described in §2.2.

system actually suffer from the increasing number of cores. For example, F2FS peaks at 18 cores, then starts to decline and ends up with a 30% performance loss. Further, in the rightmost plot of Figure 1, we observe that, even at the scale of 72 cores, most file systems do not efficiently utilize the I/O bandwidth of NVMe SSD.

The results from this experiment suggest that the performance of existing file systems are no longer bounded by the underlying device or the drive layer. Instead, the file system itself becomes the bottleneck and can not efficiently exploit the bandwidth of high performance drives.

2.2 Identifying Root Causes

Next, we investigate the CPU overhead distribution to identify the root causes of poor scalability in F2FS atop the NVMe SSDs. We use Linux performance analysis tools `perf` [8] to measure the overhead of each function call in terms of CPU cycles. We focus on four representative system calls (i.e., `write`, `create`, `unlink` and `fsync`) in a 72-core-scaling setup of F2FS. We observe that lock contention caused by unscalable data structure organization is a major source of overhead. Hence, we single out expensive lock operations, and identify their levels. Table 2 shows the overall results.

Lock cache coherence at CC level. The operations of LFS can be broadly classified into three categories: user read operations (e.g., `read` and `stat`), user write operations (e.g., `write` and `create`) and LFS internal operations (e.g., checkpoint). Most Linux file systems (FS) control the concurrency among user read operations and write operations by a relatively simple way: using a file-level inode reader-writer semaphore¹. The concurrency control among independent writes and the global checkpoint is more complicated; as shown in Table 1, LFS usually employs a traditional reader-writer lock (e.g., `cp_rwsem` of F2FS and `ns_segctor_sem` of NILFS2 [35]) at CC level to grant access for writes and checkpoint. Independent writes can concurrently update disjoint parts of the

¹In early Linux versions, the inode lock is implemented using a mutex.

FS image, and thus hold the reader-writer lock in the shared mode². As the checkpoint requires a consistent and quiescent FS image, it holds the lock in exclusive mode to prevent other writes from modifying the FS.

For `write`, we can see that more than half (50.98%) of CPU cycles are used on grabbing locks for accessing the file system. Exclusive-mode lock only allows exclusive access and thus yields expensive overhead due to serialization. Yet, global operations (i.e., checkpoint) are invoked by OS-wide `sync` syscall or periodically (e.g., 30s), which is usually less frequent, and hence do not significantly influence the throughput. The shared-mode lock, on the other hand, permits concurrent accesses, but its counter is shared among cores. As concurrent writes (i.e., shared-mode lock) are prevalent, the cache coherence on the lock counter value can be increasingly severe with the scaling, resulting in a considerable slowdown. **Serialization at IMDS level.** After entering the file system through CC level, the process needs to access and update IMDS (i.e., the in-memory indexing and data cache). Typically, LFS tends to split IMDS into different regions (e.g., inode table, inode and data) based on functionality. F2FS manages each region via a radix tree, and uses a reader-writer lock on each tree for correct concurrent execution. As shown in Table 1, for file modification operations, such as `create` and `unlink`, they require a exclusive-mode lock as they may alter the indexing. Unlike CC level, writers can be quite popular at IMDS level. With an increasing amount of concurrent writers, serialization in accessing the three radix trees becomes severe and further leads to performance drop. From Table 2, we can see that lock operations at IMDS level are the most expensive ones with 9.89% and 37.34% in total respectively, for both `create` and `unlink`.

Serialization at SA level. Finally, to persist the data blocks in a crash safe manner, the process needs to allocate space and submit the I/O requests in the correct order. LFS typically

²In this paper, to avoid confusing the FS reader/writer with the lock reader/writer, we refer to the shared-mode lock as the reader lock and the exclusive-mode lock as the writer lock.

uses only one logical space allocator to avoid overlapping allocation (i.e., concurrent writes on the same address). The space allocator uses mutually-exclusive locks for granting access. In this case, concurrent writes converge on the allocator, and wait to be scheduled in a serialized fashion before being sent to the device, which limits the overall throughput.

F2FS extends the single log schema into multi-head logging for data temperature separation. Specifically, for inode and data region, F2FS statically defines up to 3 types of temperature and employs multiple logs (6 log heads in total) on disk, each mapped to each temperature. However, from Table 2, lock contention at SA level can consume nearly half (i.e., 46.83%) of CPU cycles.

This is because the intrinsic dependencies among the temperature logs serialize the data persistence. In particular, for the crash consistency of F2FS, the data blocks must be durable before inode blocks and the file inode must be durable before the directory data blocks. As a result, in face of a `fsync`, these logs are almost processed serially although F2FS has multiple logs. Hence, the multi-head logging design of F2FS has little effect on scaling the I/O throughput.

3 MAX Design Principle

To exploit the benefits of multicore architecture and modern NVMe SSD, we formulate the following MAX design principles and describe the intuition behind them.

Principle 1: *In the CC level, using OS scheduler-assisted consensus to efficiently coordinate the external I/O requests (e.g., user writes) and internal operations (e.g., LFS checkpoint).*

LFS coordinates writes and checkpoint using traditional reader-writer lock. Recall that our study in §2.2 shows that the major overhead at CC level comes from cache coherence on the shared lock counter. Thus, a straightforward solution can be setting a local reader lock counter for each core, like scalable locks with per-core reader counters [41, 42]. A writer, to guarantee exclusive access, can simply block all further readers, and then either aggressively query the per-core counters or await until all on-going readers finish (i.e., all counters reduced to zero). While this naïve solution successfully minimizes the cache coherence among readers, the writer may either cause excessive overhead by aggressive querying or high latency due to the lazy waiting. For instance, the interrupt-processor interrupt-based aggressive query [42] is likely to interfere the latency-critical tasks on other cores, e.g., increasing the user-visible latency of read syscall. The lazy waiting approach [41] may experience periodical OS scheduling interval (e.g., 1-10 ms) and further increase the checkpoint latency. The millisecond-scale latency may be tolerable for HDD and SATA SSD, but is unacceptable for NVMe SSDs with ten to hundreds of microsecond latency.

On the other hand, an outstanding advantage of the kernel FSes is that they run on the OS control plane. This has not

been exploited to assist concurrency control of LFS. For example, to guarantee process fairness, the process in the kernel mode frequently invokes the OS scheduler for scheduling. A typical case is the exit of FS syscall, implying the end of I/O.

MAX uses a new reader-writer semaphore namely Reader Pass-through Semaphore (RPS) for concurrency control. The RPS uses per-core counters to reduce cache coherence overhead, and introduces *scheduler-assisted consensus* to coordinate the external and internal operations with less overhead.

Principle 2: *In the IMDS level, the IMDSes are partitioned by the file inode ID for concurrent file-level in-memory indexing and caching. Further, the IMDSes of the same file are repacked to avoid page-level (e.g., 4 KB) false sharing, so as to facilitate the concurrent file-level persistence functions.*

Earlier in § 2.2 we showed that serialization of accessing indexing trees at IMDS level is expensive. A feasible way to accommodate concurrent writers is to split the trees into multiple ones, similar to the non-volatile main memory FS [57]. Partitioning the memory-based storage system is relatively flexible due to fine-grained access granularity (e.g., byte-scale) of the main memory. However, such a partition method can not be directly applied to block-based SSD FS due to different and coarse access granularity. Traditionally, the FS for block storage often stitches small-sized file metadata and file system metadata into a shared block, which introduces extra contention. Therefore, scaling the IMDS while delivering flash- and concurrency-friendly on-disk layout remains a challenge for MAX.

In MAX, we propose a new IMDS abstraction, *file cell*, to repack data and metadata to allow multiple indexing entities, thereby lowering the chance of serialization. Additionally, we realize SSD- and concurrency-friendly on-disk format for file cell by setting a dedicated page for each inode and stitching the small unaligned flushes as pages. In this way, MAX can access the IMDS and write the buffered pages to the persistent storage with less contention.

Principle 3: *In the SA level, the drive space is divided into multiple independent logs; independent file operations can allocate space and be distributed to minor logs concurrently. The ordering and consistency of dependent file operations among multiple logs are delegated to crash recovery.*

Our study in §2.2 shows that having only one space allocator for each type of data can introduce considerable overhead for concurrent writers. MAX addresses this issue by partitioning the log space, like other log-based storage systems [10, 57]. Log partitioning introduces challenges to maintain the concurrency and crash consistency over multiple minor logs (mlog).

For concurrency control, leveraging the file interfaces and semantics of FS, MAX distributes complete file operations (not simply data blocks) to a mlog. In other words, when persisting files (e.g., `fsync`, or `write` a file with `O_SYNC` flag), MAX submits the data blocks that need to be persisted atomically to the same mlog. This avoids concurrency control over multiple mlogs and brings higher concurrency.

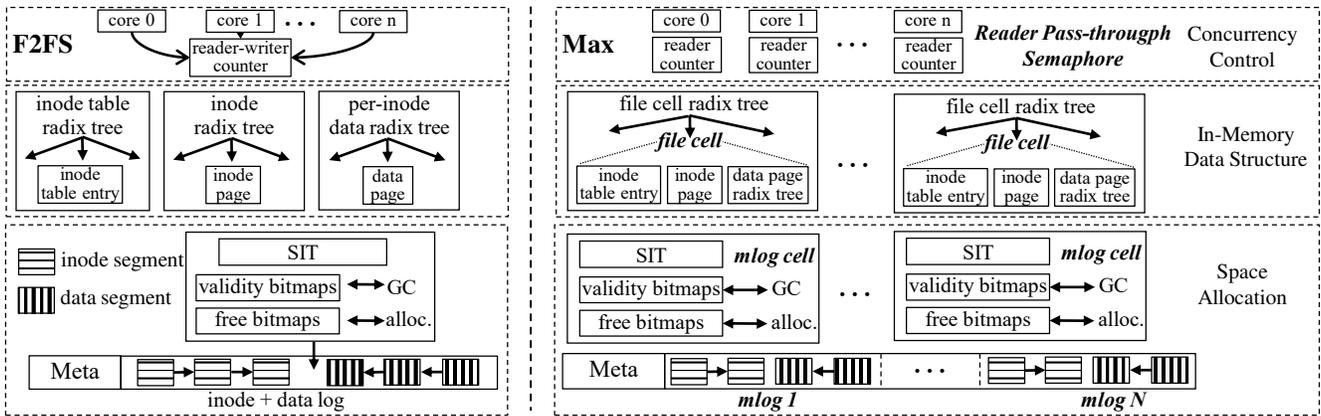


Figure 2: A comparison of F2FS and MAX. MAX introduces Reader Pass-through Semaphore (RPS) (§4.1) at CC level, file cell (§4.2) at IMDS level and mlog cell (§4.3) at SA level for higher concurrency.

For crash consistency, each mlog keeps its localized consistency just like traditional LFS. In MAX, the same file of different versions can be distributed to different mlogs. MAX embeds a global version number in each inode block to record the ordering across multiple independent mlogs, and finds the newest file during crash recovery using the global version.

By persisting independent file operations to independent mlogs and delaying the persistence ordering to recovery, MAX enables highly concurrent persistent functions.

4 MAX Implementation

We implemented MAX by modifying F2FS. Figure 2 shows a side-by-side comparison between MAX and F2FS.

4.1 Reader Pass-through Semaphore

MAX replaces the traditional reader-writer semaphore of F2FS (i.e., `cp_rwsem`) with Reader Pass-through Semaphore (RPS). We use Figure 3 (referred by arrow numbers) and Algorithm 1 (referred by line numbers) to elaborate the RPS control flow. **Concurrent readers.** RPS borrows the idea of per-core reader lock counter. With a private counter for each core, concurrent readers can independently increase or decrease the counter value without cache coherence from different cores (lines 1-8). The major overhead (i.e., 50.98% in Table 2) at the CC level is thus removed.

Exclusive writer. RPS introduces a “Scheduler Free Rides” mechanism to avoid high overhead and latency for exclusive-mode lock. The key idea is to leverage the CPU scheduler to efficiently check the counter value of each core. The original design goal of the CPU scheduler is to coordinate processes, which lets it frequently access the cores. Scheduling itself searches several queues, so adding RPS logic (i.e., check the reader counter) atop it costs extra little. Hence, the counter values of different cores are frequently retrieved with low overhead, thereby taking the free rides of the scheduler.

Specifically, the pending writer first locks `wsem` (i.e., Linux writer semaphore) on each core to block all further incoming readers and writers (line 15, arrows ① and ④). The writer sets the per-core notification flags, and then goes to sleep and waits for all flags cleared (i.e., readers on all cores have left the critical section, lines 16-19). Next, the on-going readers continue as usual except the last reader on each core finishes by yielding the execution to the CPU scheduler (lines 9-10, arrow ②). The CPU scheduler then clears the per-core notification flag, which indicates that the on-going readers of that core are all finished. For cores with no on-going readers, the RPS utilizes the opportunity of kernel preemption to let the scheduler check the counter value and clear the notification flag (lines 27-28, arrow ⑤). With all notification flags cleared, the scheduler can then wake up the writer and let it start to execute (lines 18-19 and 29, arrow ⑧).

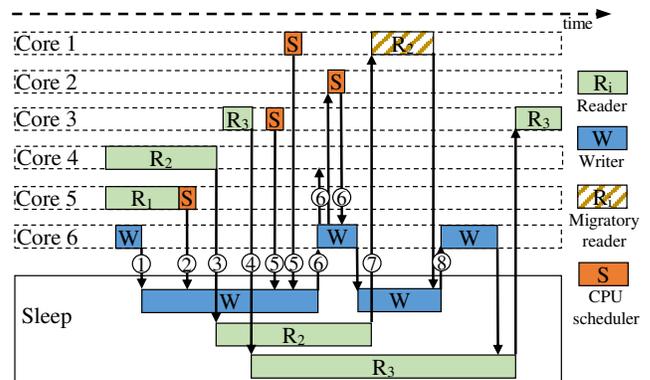


Figure 3: “Scheduler Free Rides” mechanism of RPS. Described in §4.1.

Corner cases. In our original RPS design, there are two corner cases: late preemption and task migration. First, in the “Scheduler Free Rides” mechanism, the scheduler relies on opportunities of kernel preemption to check the reader lock counter on zero-reader cores. While preemptions are usually frequent, a worst-case scenario can take up to 10 ms (i.e., tick

Algorithm 1: RPS Pseudo-code

```
1 def read_lock (rps):
2   while rps.wsem ≠ FREE do
3     wait(rps.wsem == FREE) /* woken up by line 25 */;
4     this_cpu_inc(rps.percore_reader);
5 def read_unlock (rps):
6   readers ← this_cpu_read(rps.percore_reader);
7   if readers > 0 then
8     readers ← this_cpu_dec_return(rps.percore_reader);
9     if rps.wsem ≠ FREE and readers == 0 then
10      yield();
11  else if readers == 0 then /* migration reader exists */
12    if atomic_dec_test(rps.migration_cnt) then
13      wake_up_writer();
14 def write_lock (rps):
15   lock(rps.wsem);
16   for core ∈ get_online_cpu() do
17     per_cpu_set(rps.noti_flag, core, 1);
18   for core ∈ get_online_cpu() do /* wait for all flags cleared */
19     wait_timeout(per_cpu(rps.noti_flag, core) == 0, TIMEOUT);
20     if timeout then
21       cores_unfinished ← check_notification_flags();
22       send_ipi(cores_unfinished);
23   wait(atomic_read(rps.migration_cnt) == 0) /* woken up by line 13 */;
24 def write_unlock (rps):
25   unlock(rps.wsem);
26 def schedule (core, task):
27   if this_cpu(rps.percore_reader) == 0 then
28     this_cpu_set(rps.noti_flag, core, 0);
29     wake_up_writer();
30 def migrate_task (task, src, dst):
31   /* Tasks holding RPS are off the src and dst CPU now, so it's safe to
32    modify the per-core reader counter */;
33   per_cpu_dec(task.rps.percore_reader, src);
34   atomic_inc(task.rps.migration_cnt);
```

preemption interval) for the scheduler to check all counters. This delay is unbearable for high performance SSDs and can cause a long writer latency. To handle this, RPS sets the writer to wake up periodically. Then the writer actively invokes inter-processor interrupts to check reader counters of unfinished cores (lines 20-22, arrow ⑥). Note that the wake-up interval is configurable in RPS (100 us by default).

The second corner case is the task migration. In the multicore execution environment, a reader on one core can be migrated to another core (arrows ③, ⑦). Therefore, RPS sets up a global migration counter. Upon task migration, the RPS decreases the local counter of source core by one and also increases the global migration counter by one (lines 31-33). The migrated reader therefore decreases the global migration counter instead of the local one when finishes (lines 11-12). Thus, the pending writer needs to wait until both local counters and the migration counter all decreased to zero before accessing (line 23, arrow ⑧).

4.2 File Cell

As shown in Figure 2, MAX organizes the IMDSeS using the file cells and indexes them by multiple trees. This subsection describes the indexing and the format of file cell in details.

File Cell indexing. Each file cell encompasses the inode table entry, inode page³, and data page of a single file. MAX then divides the file cells into multiple groups and indexes each group using a radix tree. MAX places each file cell to a tree by hashing the inode ID of the file (i.e., inode ID modulo the number of trees). Each radix tree accepts the inode ID as key and outputs associated file cell. Note that the number of indexing entities (i.e., radix trees) is configurable. Having more trees yields a lower chance of serialization but can also lead to high memory consumption. We set the number as half of the number of cores as we observe that setting more trees beyond that does not lead to better performance (see § 5.4 for details). Thus, MAX lowers the chances of serialization as well as the number of indexes.

File Cell data format. MAX uses a dedicated page for the inode of each file. To reduce memory consumption, we use the following approach. If the unaligned data can fit in the inode page, MAX appends that to the end of the inode page. If not, the inode owns the entire page. For example, consider a file with 6 KB data and a 256 B inode. MAX sets two pages for that file cell. The first one is a 4 KB data page. The second one is the inode page that contains the 256 B inode plus the 2 KB unaligned data.

Unlike the journaling FS that inodes are placed in a pre-determined location, inodes of LFS are updated in an out-of-place manner, thereby forcing the inode table to be placed in a fixed on-disk location for indexing. Additionally, information similar to the inode table entries must be persisted simultaneously to locate the newest inode. However, the inode table entry is extremely small (e.g., 9 B). This brings challenges for LFSes to achieve high concurrency, low write amplification without breaking the fixed-location property.

MAX realizes high concurrency and persistence of inode table entries using two representations. For in-memory representation, with the byte-addressability of DRAM, MAX partitions the inode table, and distributes the inode table entries to each file cell. All operations, except checkpoint, access or modify the inode table entries inside the file cell only in the memory. Second, we reuse the classic representation (i.e., compacting different entries in pages) on disk to guarantee entries are always stored in the pre-determined locations. We rely on the periodical checkpoint to persist entries. Note that, in this design, updates on inode table entries since the most recent checkpoint can be lost in the face of a sudden crash. We further discuss how MAX uses roll-forward recovery to reconstruct inode table entries in §4.4.

4.3 Mlog

MAX extends the multi-head logging of F2FS by splitting the larger log into minor logs (mlogs), as shown in Figure 2. Each mlog keeps the data and inode logs each with up to three

³In this paper, we use the commonly-used inode table entry and inode to refer the specific node address table (NAT) and node structure of F2FS.

temperatures as in F2FS. The number of mlogs is configurable. Ideally, the number of mlogs can be the same as the number of cores to achieve the highest concurrency. However, for small capacity drives, having too many mlogs makes it hard to accommodate large files due to limited capacity per mlog. We further evaluate and discuss how to choose the appropriate number of mlogs in §5.4.

Mlog cell. To support mlog, MAX splits the allocation-related IMDSes, forming individual mlog cells. Following the design of F2FS, MAX uses segment info table (SIT), validity bitmaps and free bitmaps for space allocation. Yet, different from F2FS, MAX splits and co-locates the table with the two kinds of bitmaps in mlog cells, as shown in Figure 2.

The SIT maintains the validity of all blocks for the corresponding mlog. The free bitmap records the free 4 KB blocks. Upon data flushing (e.g., fsync), MAX selects a mlog cell in a round-robin fashion and searches the free bitmap for free blocks. Then, MAX sends the data to the corresponding mlog.

Garbage collection. Each mlog cell performs garbage collection (GC), i.e., victim selection and block identification and migration, independently. As GC performs at the granularity of the *section* (consecutive 2 MB *segments*), MAX keeps the validity bitmaps to record both the dirty segments that need GC, and the valid blocks per segment in the section.

For each mlog cell, MAX uses existing victim selection policies [32, 52] and the slack space recycle (SSR) [37] technique of F2FS; MAX always performs GC for inode log and we explain this in §4.4 by Figure 5. Since the data/inode blocks can be spread across different mlogs, MAX identifies not only the valid block in the mlog, but also the freshness of that block among all mlogs, by comparing the address of the block (in the mlog) with the newest address of the file data/inode; MAX migrates only the valid and newest block. If a space allocation can not find enough space in all mlog cells (e.g., the used disk volume is higher than 95%), MAX turns back to the single logging as in F2FS. While, in this case, the SA level concurrency is sacrificed, MAX avoids severe fragmentation.

4.4 Consistency

Concurrency Consistency. In traditional Linux file systems, the concurrency consistency is mostly handled by the Virtual File System. The actual file system only needs to lock on the target file(s). Therefore, with VFS remains intact, MAX simply locks on the file cells of the target files to ensure correct execution order.

Crash Consistency. After a crash (e.g., power outage), MAX recovers the state by the following two steps: (1) roll back to the latest consistent checkpoint; (2) perform roll-forward recovery on all mlogs.

Here, we use an example in Figure 4 to present the roll-forward of MAX. First, for each mlog, starting from the latest checkpoint, MAX rebuilds the inode log and forms lists of inode blocks (b.1). Next, MAX merges the per-mlog inode list.

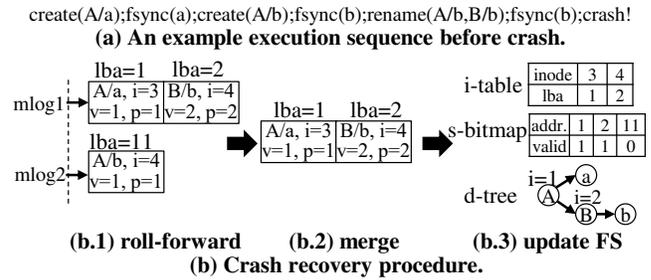


Figure 4: An example of MAX’s crash recovery. *lba*: logical block address, for recovering space bitmap (*s-bitmap*); *i*: inode number, for recovering inode table (*i-table*); *v*: global version, for merging; *p*: parent’s inode number, for reconstructing the FS directory tree (*d-tree*). Described in §4.4.

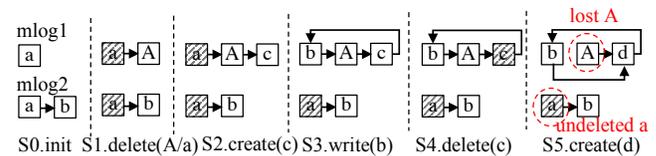


Figure 5: Performing inode GC in SSR mode.

Recall that MAX uses round-robin placement to writes. Therefore, outdated versions of an inode may still exist. For example in Figure 4(a), the file *b* is modified twice; `create(A/b)` is located in *mlog2*, and `rename(A/b, B/b)` is distributed to *mlog1*. Such different versions can bifurcate the merged inode list. To overcome this, MAX embeds a global version number (*v*) inside the inode block during each flush, and uses it to identify the latest inode block during recovery (b.2) if needed. Finally, MAX updates the inode table, the space bitmap and the directory tree with the merged inode list (b.3).

The inode list can also be affected by the GC policy. For example, using SSR, a state-of-the-art space allocation policy, can lead to an inode loop, causing consistency issues. Specifically, in Figure 5, the user deletes file *a* in directory *A*. Now, the inode of *a* is invalidated, and its parent inode *A* is updated (S1). Later when there are no more clean segments, MAX switches to SSR mode and directly reuses the obsolete blocks (e.g., *b* of S3). In this case, an inode loop is formed, and a further inode update may bifurcate original inode list, resulting in updates loss (S5). In S5, the latest *A* is lost, and the deletion of *a* is lost as well. During next roll forward, the invalid *a* in *mlog2* would be considered valid. Therefore, in SSR mode, MAX still uses common GC for the inode log.

4.5 Other Important Implementations

This subsection describes how MAX’s approach retains other aspects of the design of F2FS while improving concurrency.

Extent cache. F2FS uses bitmaps to record the addresses of the data blocks for a file. A bitmap-based approach is efficient for lookups of a specific point in a file but unfriendly when scanning continuous ranges of addresses. Thus, F2FS uses

per-inode extent cache to speed up address lookup (esp. range lookup). However, the per-inode extent cache is indexed by a global radix tree (`extent_tree_root`) protected by a mutex (`extent_tree_lock`), similar to the single inode radix tree. MAX splits the extent cache to each file cell for concurrent extent cache access.

Inode table journal. As we mentioned in §4.2, the inode table entry of F2FS is extremely compact (9 B). Directly updating the inode table entry to its original location is likely to incur many small I/Os, which is not friendly to performance and lifetime of flash storage. Therefore, F2FS employs an inode table journal in the spare space of the Meta region. To quickly find the dirty inode table entries that need to be journaled, F2FS uses a global linked list guarded by a spinlock (`nat_list_lock`). This introduces significant contention on the shared list when the number of threads is large. MAX only links the inode table entries for each file cell group to alleviate the contention on the list. During checkpoint, MAX scans the per-group linked list to generate the inode table journal.

Resource counters. Similar to many FSes, F2FS uses delayed allocation techniques to postpone resource allocation until the data blocks are finally sent to the persistent storage. F2FS uses many resource counters (e.g., `total_valid_block_count`) to pre-reserve FS resources for incoming I/Os. These counters are shared globally under the protection of a spinlock (`stat_lock`), and become scalability bottlenecks in the multicore environment. Actually, the FS only requires the approximate value of these counters, i.e., only needs to determine whether the incoming request fits in the FS. Hence, MAX replaces these counters with scalable approximate counters (`percpu_counter` [33]).

These modifications made by MAX do not change the write ordering, the consistent metadata format or crash recovery logic, and thus do not impact consistency.

5 Evaluation

We first evaluate MAX against state-of-the-art Linux file systems on file operations (§5.1) and applications (§5.2). Then, we perform experiments of MAX under high volume utilization (§5.3). Next, we study the performance contributions of individual design aspects of MAX (§5.4). Finally, we examine the memory consumption by running MAX (§5.5).

Testbed. The testbed is equipped with 4 Intel Xeon Gold 6140 CPU processors; each CPU has 18 physical cores (totally 72 cores) running at 2.30 GHz. The platform has 250 GB DRAM, but only 10% DRAM (i.e., 25 GB, the Linux default configuration) is used for page cache. The experiments in this section are performed on a flash-based Intel DC P3700 SSD, whose performance is presented in Table 3.

File systems setups. We compare MAX with four Linux file systems (ext4 [44], SpanFS [29], XFS [18] and F2FS [37]) in Linux kernel version 4.19.11. Ext4 and XFS are popular journaling FSes used by many Linux distributions and storage

Type	Model	Seq. Bandwidth	Rand. IOPS
NVMe	Intel DC P3700 2TB	Read: 2800 MB/s	Read: 450K
		Write: 1900 MB/s	Write: 175K

Table 3: SSD Specifications.

backends. SpanFS is a recent scalable journaling FS built on Ext4. We set the number of the parallel journals of SpanFS to 72, the same as the number of physical cores. All tested FSes are mounted with default options. The numbers of file cell radix trees and mlog cells are set to 36 and 8 respectively. For upper bound comparison, we use an alternative MAX-mem by disabling the `fsync` and page cache writeback functions of MAX to avoid duplicated copy.

Workloads. FxMark [48] is used to test multicore scalability. FxMark concurrently and repeatedly executes individual file operations or application processes. All tests last for at least 30 seconds and issue over 50 GB data.

5.1 Microbenchmark

5.1.1 File Operations Evaluation

Overwrite. Figure 6a shows the results of DWOL workload of FxMark. MAX achieves nearly 56× speedup at 72 cores for overwrite operations. MAX outperforms F2FS and SpanFS (the second-best) by 35× and 2× at 72-cores respectively. The key contributing factor to MAX’s overwriting performance is the RPS. Overwrite operations are performed in parallel by updating individual data pages and hence do not frequently trigger page cache flushes. Thus, the major overhead occurs at the CC level. While F2FS is bounded by the CC level reader cache coherence, MAX achieves high concurrency with per-core counters in the RPS. SpanFS and XFS use multiple in-memory journal buffers, and thus serve concurrent overwrites in parallel. Nonetheless, the journaling process adds extra overhead compared to MAX.

Append write. Figure 6b reports the results of DWAL workload of FxMark. We find that MAX delivers the best performance, and achieves almost 2× the throughput of F2FS. The major contributing techniques here are the file cell and the mlog. As append writes quickly fill the page cache and trigger flushes, the I/O becomes the major bottleneck. Specifically, append writes require new data pages and new inode pages, resulting in insertions into the indexing trees. Such operations of the traditional FS incur frequent serialization at IMDS level. MAX reorganizes the IMDS with file cells and uses multiple indexing trees. Hence, the chance of serialization becomes significantly lower. Moreover, at SA level, MAX allocates space from individual mlog cells, and flushes new blocks to individual mlogs. In F2FS, the NVMe SSD is however underutilized due to the single sequential log allocation and access.

File creation. Figure 7a presents the results of MWCL workload of FxMark. We observe that, on file creation operations at 72 cores, MAX achieves 2.8× higher performance than SpanFS (the second-best) and 18.6× higher than F2FS. File creations

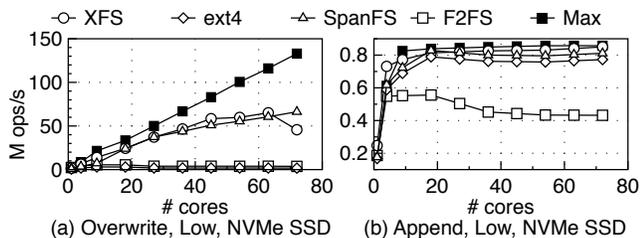


Figure 6: **Data scalability with FxMark.** ((a): overwriting blocks of private files, (b): appending blocks to private files.)

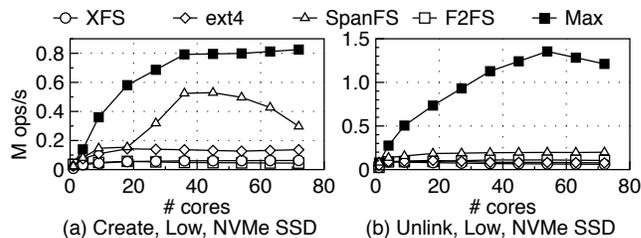


Figure 7: **Metadata scalability with FxMark.** ((a)-(b): creating or deleting empty files in private directories.)

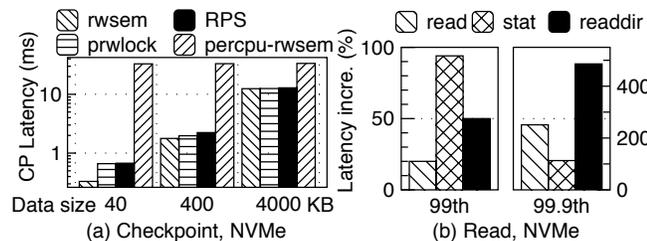


Figure 8: **Checkpoint and read latency with different lock techniques.** (a): Checkpoint latency with different data size and locks. (b): Read long tail caused by IPIs.

mainly consist of three steps: allocating file inode, growing directory data/inode blocks and writing back data/inode page. MAX scales well in all steps. First, file inode allocation in MAX includes the file cell allocation and insertion. MAX employs a per-core inode ID allocator and multiple file cells radix trees, thereby avoiding contention on the inode and file cell allocations; multiple radix trees also lower the chance of serialization at insertion operations to the indexing. Second, directory data and inode block grow concurrently in all tested file systems without hurting the concurrency. Third, when dirty pages are evicted from the page cache, mlog cells of MAX enable the threads to allocate space concurrently, and to distribute the dirty inode blocks to individual mlogs.

File deletion. Figure 7b shows results of MWUL workload of FxMark. MAX achieves $11.5\times$ and $6.1\times$ performance at 72-cores against F2FS and SpanFS (the second-best) respectively. The reasons for MAX's good scalability on file deletion are similar to that of file creation. In MAX, directory entries and inode pages are truncated independently in file cells. Also, mlog cells reclaim disk space in parallel. We observe that MAX's throughput declines since 54 cores. Further analysis suggests that the root cause is the page cache lock contention (i.e., `i_wb_list`) from VFS.

Checkpoint. We replace traditional `rwsem` in F2FS with several alternatives, and then collect the latencies when checkpointing variable-sized data. Figure 8a shows the results; we observe that `prwlock` [42] and RPS hardly affect the checkpoint latency. However, Linux `percpu rwsem` slows the checkpoint significantly, as its exclusive-mode lock requires RCU-based quiescence detection, where all cores have done a context switch and executed a full memory barrier.

File read. We co-run multiple foreground tasks, i.e., reading

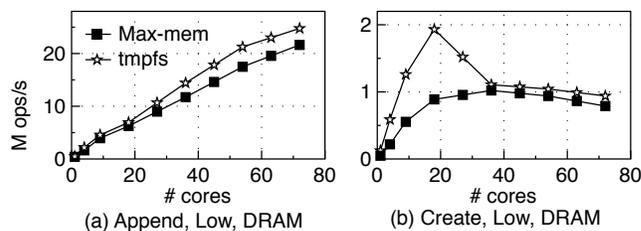


Figure 9: **Upper bound evaluation with FxMark.** ((a): appending blocks to private files, (b) creating empty files in private directories.)

files (read), listing file attributes (`stat`), reading directories (`readdir`), which are conflict-free and pinned to cores, with a background task which triggers checkpoint periodically. We then re-run the above scenario without the background task and compare the results. We find that the foreground tasks are almost unaffected by `percpu rwsem` or RPS. However, we observe that the foreground tasks are susceptible to the inter-processor interrupts (IPIs) of `prwlock`. Figure 8(b) shows a performance decline: the 99.9th latency increases by up to 486%. The reason behind the long tail is that these read-dominant tasks mostly complete in a short time by hitting cache or accessing memory, which are easily affected by the forced context switch caused by IPIs.

Comparison with SpanFS. Through the aforementioned tests, we note that SpanFS scales well on single-file operations (i.e., overwrite, append write), but yields suboptimal performance on multiple-files operations (i.e., create, unlink). This is because of the global consistency maintenance across multiple journaling services. For instance, the newly created files can be distributed to a different journaling service from its parent, and the connection between the new file and its parent's directory entry (`dentry`) must be established which is quite expensive. MAX, due to the out-of-place update nature of LFS, can directly dispatch the newly created file along with the `dentry` to an arbitrary free mlog.

5.1.2 Max-mem Performance Evaluation

We use MAX-mem to measure the performance improvement led by a sufficiently large bandwidth (i.e., using memory as backend). For comparison, we adopt `tmpfs`, a simple wrapper of VFS, as the theoretical upper bound [48]. Figure 9a reports

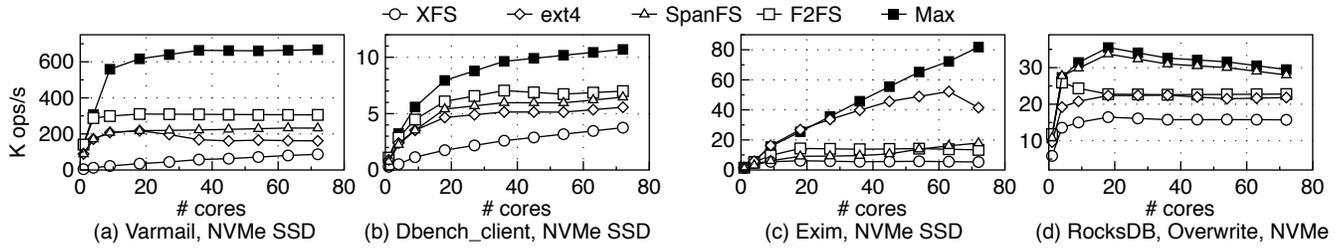


Figure 10: **Macrobenchmark.** The workloads are write-intensive and stress underlying device. Described in §5.2.

Workload	Average file size	# of files	Directory width	I/O size	R/W ratio
Fileserver	128 KB	10K	20	1 MB	1:2
Varmail	16 KB	1K	1M	1 MB	1:1
Dbench	client.txt, default configuration				
Exim	split spool directory, smtp_accept_max = 500				
RocksDB	overwrite, value_size=8k, disable compression				

Table 4: **Application workload characteristics.**

the results of append write operations. The throughput of both Max and tmpfs scale linearly. The little gap is caused by the overhead of supporting block storage and pre-checking the availability of FS resources (e.g., the number of data blocks).

Figure 9b indicates that the throughput of file creations of Max-mem comes close to tmpfs. Max-mem does not continue to scale mainly due to the VFS-wide spinlock `s_inode_list_lock` that serializes new inode insertions into the `s_inodes` list. We also notice the huge gap between MAX-mem and tmpfs at 18-core. Tmpfs only stores in-memory states. While in MAX, the dentry and inode also contain information for on-disk states, such as the dentry hash table and data block indexes. This introduces significant costs for create operation. However, when the VFS-wide spinlock kicks in, the gap becomes much smaller.

5.2 Macrobenchmark

We use Filebench [46], Dbench-client [7], RocksDB [22] and Exim [1] from FxMark to evaluate Max’s scalability under applications workloads. Table 4 summarizes the characteristics of these workloads. The main rationale for choosing these four workloads is that they are both write- and I/O-intensive and can thus stress the multicore scalability.

Varmail. Varmail contains frequent metadata operations and `fsync`. Figure 10a shows the results of Varmail. Max outperforms SpanFS by 2.9× and F2FS by 1.1×. For independent file operations, MAX updates the in-memory file cells concurrently with 36 indexing groups. When `fsync` is invoked, Max chooses a free mlog cell and persists the inode pages and data pages of the file cells. In contrast, F2FS uses only three shared radix trees and need to serialize concurrent threads for space allocation. Notably, SpanFS performs even worse than F2FS due to its inefficiency in handling the file creation and deletion followed by a `fsync`.

Dbench-client. Figure 10b shows the results of Dbench. Max

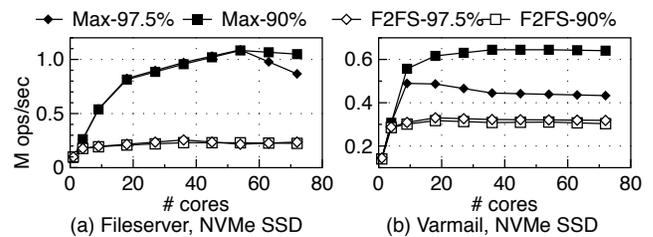


Figure 11: **Performance under high disk volume utilization.** The number next to the file system is volume utilization.

performs best among all tested file system. F2FS stops to scale at 36 cores while MAX continues to. This is because the Dbench-client performs a sequence of read, write, create, unlink, stat, rename and sync operations. For non-durable operations, such as write, create and unlink, MAX delivers higher concurrency by executing operations inside each cell.

Exim. Exim focuses on small files creation and deletion. Here, we use the scalability-friendly version of Exim from the FxMark, where create and delete are performed almost in private directories. However, Figure 10c shows that only MAX scale well in this modified version, outperforming F2FS by 6×. The reason for MAX’s good scalability is the same as file creation and deletion in §5.1.1.

RocksDB. RocksDB introduces multi-threaded flush and compaction to boost performance. We use `db_bench`, which runs four client threads to put keys in a single RocksDB instance atop a native FS. Then, we increase flush and compaction threads up to 72 threads. Figure 10d shows the result. MAX outperforms its peers. Note that the throughput of MAX starts to decline after 18 cores. We assume this can be caused by the following three reasons. First, RocksDB frequently invokes durability functions and hence occupies a large fraction of the device bandwidth, lowering throughput of user requests. Second, scaling over 18 cores (maximum cores of a single CPU) incurs Non-Uniform Memory Access and PCI bus routing. This throttles IOPS due to the inefficiency of remote access. Third, RocksDB manages all its files including metadata files and SSTable files under the same directory, causing a medium level contention on the shared directory.

5.3 High Volume Utilization Evaluation

We evaluate the performance under high file system volume utilization and high GC overhead in this subsection. We for-

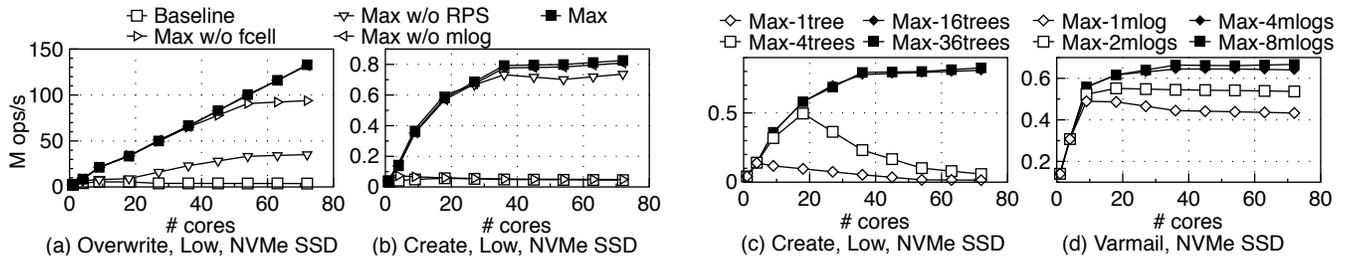


Figure 12: **Performance contributions.** Described in Section 5.4.

mat the 2 TB NVMe SSD and create a 200 GB partition for the test. This ensures that the device GC rarely occurs during the test. After filling up the tested file systems with two utilization ratios (90%, 97.5%), we issue another 200 GB of random overwrites to make the file system further fragmented.

Figure 11 plots the results of Fileserver and Varmail workload with FxMark. At 90%, MAX outperforms F2FS by 4.4× and 2.1×, in Fileserver and Varmail, respectively. In this case, both MAX and F2FS switch to the SSR mode. In SSR mode, besides concurrent access to file cells, MAX still offers concurrent space allocation. However, in this case, the inode updates need cleaning. The serialized checkpoint after each cleaning limits the further performance increase of MAX.

The 97.5%-utilization MAX, at 72-cores, outperforms F2FS by 3.5× and 36% in Fileserver and Varmail, respectively. We observe that the throughput of MAX starts to decline after 18 cores in Varmail. This is because MAX regresses to single logging under high volume utilization ratio. In this scenario, as RPS and file cell continue to contribute, MAX survives greater performance drop due to the single log access.

5.4 Understanding the Performance

We individually analyze the contribution of *file cell*, *mlog* and *RPS* to the performance of MAX. We setup MAX with different configurations (the left half of Figure 12): (1) F2FS (baseline), (2) MAX without file cell (Max w/o fcell), (3) MAX without mlog (Max w/o mlog), (4) MAX without RPS (Max w/o RPS), (e) full-fledged MAX (MAX). The tests are the same as that in §5.1.1 and §5.2, i.e., DWOL, MWCL and Varmail.

Figure 12a-d show that, compared to the baseline, all three techniques improve the performance. Specifically, RPS has a greater boost on overwrite operations, as heavy cache coherence takes up to several milliseconds but in-memory updates only cost hundreds of nanoseconds. The file cell is more effective for complex create operations as modifying IMDS is (e.g., inode initialization and hash calculation) expensive. Mlog cell does not exhibit significant impact on Figure 12a-b because these tests are not I/O intensive.

We also study the sensitivity of MAX to the number of IMDS radix trees and the number of mlogs. Figure 12c-d plot the results of varying the number of trees and mlogs. We choose create operation for trees as it frequently modifies the indexing structure, and Varmail for mlogs due to its heavy

fsync. We observe that MAX does not gain improvement after more than 36 trees. The major reason is that the bottleneck, after 36 trees, has shifted from file system to the VFS. For write intensive workload, 8 mlogs have almost saturated the throughput. This is because MAX, following the design of F2FS, performs checkpoint to shrink cached entries (e.g., inode table, inode). In current single-threaded checkpoint design that blocks all write operations, assigning each CPU two logs in our platform is enough currently.

5.5 Memory Consumption

We examine the extra memory consumption introduced by deploying MAX. We measure the peak memory usage for each workload when running 72 processes. Table 5 reports the result. The memory usage is categorized into two sets: (1) static: memory used by the data structures of each file system, and (2) cache: memory used by page cache. We find that MAX does not introduce much memory overhead for static memory use. At CC level of MAX, the single RPS keeps two per-core counters. At IMDS level, MAX has to maintain 36 file cell radix trees. At SA level, the segment info table and all bitmaps are physically partitioned and distributed to mlog cells, which requires no extra memory. Hence, we assume that the extra static memory usage from three levels is acceptable.

For page cache, MAX consumes more memory than F2FS at peak. This is because with file cell, the dirty pages aggregate faster to the page cache (i.e., increase memory consumption). Meanwhile, the mlog cells can also write back dirty pages faster (i.e., decrease memory consumption). As a result, extra page cache memory consumption is tolerable.

Workload	Static (MB)			Cache (MB)		
	F2FS	MAX	Incr.	F2FS	MAX	Incr.
Varmail	24.90	24.97	0.3%	1411	1484	5.2%
Dbench	24.90	24.97	0.3%	310	345	11.3%
RocksDB	24.90	24.97	0.3%	15	18	20%

Table 5: **Peak memory consumption during workload execution.** Static: data structures, cache: page cache.

6 Related Work

File system scalability study. Multiple studies have discussed the scalability issues in the FSES [14, 15, 17, 48]. Fx-

Mark [48] argues that file system scalability does not depend much on the storage media (i.e., Figure 1 of [48]) which is different from our observation in §2. The root cause is that, in FxMark, file operations are performed in memory with no `fsync`. Yet, when cache has little effect on absorbing I/O traffic, file system scalability *does* depend on the storage media. Such scenarios include: (1) file operations in Direct I/O mode (e.g., QEMU none cache mode [3]); (2) applications contain frequent `fsync` or `fdatasync` calls (e.g., SQLite [5]); (3) frequent cache eviction (e.g., high memory pressure).

File system scalability improvement. Another group of prior work focus on improving the file system scalability. Commuter [17] and ScaleFS [11] employ scalable software design by connecting scalability to interface commutativity. Both Commuter and ScaleFS target on a more scalable kernel (i.e., `sv6`) with relaxed POSIX semantics. MAX studies the multicore scalability on mature, widely-used Linux and POSIX interface. Park *et al.* [50] and Son *et al.* [56] improve the scalability of journaling. MAX focuses on LFS instead.

File system partition. SpanFS [29] and IceFS [43] partition the journaling FS. SpanFS distributes files and directories under each *domain* for scalability. IceFS partitions the file system into directory subtrees called *cubes* for failure isolation. Unfortunately, IceFS and SpanFS can incur significant overhead from maintaining the global hierarchical namespace (e.g., sharing a single physical journal in IceFS and coordination across journals in SpanFS). MAX partitions the file system at the granularity of file operations, where the file operations to `mlogs` are totally independent of each other.

Scalable NVMM FS. Some non-volatile main memory (NVMM) file systems [36,57] employ per-core or per-process data structures, which seem to be the direct solutions to SSD-based FS. The major difference of the scaling SSD-based and NVMM-based FS lies in the access granularity. NVMM FS can partition the FS at a finer granularity (e.g., 8 bytes); for example, NOVA [57] atomically updates the 8 B inode pointer to ensure the persistence and consistency of a file write operation. Such fine-grained partitioning is unfriendly to SSD FS with block access granularity (e.g., 4 KB). If MAX directly isolates the inode pointer (i.e., inode table entry) for each file and directly persists each pointer to SSD, MAX would suffer from huge write amplification and extra PCIe round trips. Hence, MAX introduces the file cell to repack the data structures to align the block granularity as well as to scale the file-level performance.

For concurrent space allocation, NVMM FS can partition the drive space with finer granularity, e.g., per-file log. Such fine-grained partitioning is unnecessary, or even inefficient for SSD FS. Fine-grained partition trades scalability for fragmentation. In MAX, we take a sensitivity study to find the appropriate number of logs in §5.4. Further, NVMM FS such as NOVA uses a journal to maintain consistency over multiple log partitions. MAX takes a different and more SSD-friendly approach: distributing the atomic operation to a coarse-grained log par-

tion without spanning. To maintain the consistency over multiple log partitions, MAX embeds a global version number in each inode block instead of using a journal.

Scalable lock designs. Myriad work [16, 19, 21, 30, 31, 34, 39, 41, 42, 45, 47] devise scalable locks. One category [19–21, 30, 31, 39] employ distributed reader indicators (e.g., per-NUMA node reader indicators) or similar designs to reduce the cache coherence traffic across NUMA nodes or CPU cores in the common case, which is similar to RPS. RPS uses per-core reader indicators for optimal reader-reader scalability, and uses a different “scheduler free rides” mechanism to soften the impact on the exclusive-mode lock. This mechanism aims to leverage the CPU scheduler to increase the responsiveness of the lock writers while reducing the impact (e.g., forced context switch) on other CPU cores, which we found very effective for the LFS concurrency control. RCU [47] and its extensions [34, 45] allow readers just for read-only traversals. Hence, they can not be directly used in the LFS concurrency control, as readers need to perform updates. RPS maintains the number of readers in per-core reader counters, the same as in [16, 41, 42]. RPS differs from them in the writer procedure. Before updating protected data structures, the writer of Linux `percpu rwsem` [41] must wait a significantly long grace period [42]. The writer of `prwlock` [42] actively broadcasts IPIs to check reader status, which causes unnecessary context switches of the on-going readers. RPS leverages the CPU scheduler to retrieve reader status; the last reader of RPS voluntarily yields cores to the CPU scheduler, which enables the writer to check readers efficiently without affecting readers.

7 Conclusion

The bandwidth of SSDs has been surging over the last decade. However, through a performance study, we notice that modern Linux file systems do not offer enough multicore scalability and hence can not fully exploit the abundant bandwidth of high performance drives. We propose MAX, a multicore file system to effectively alleviate the lock contention of the file system. MAX introduces reader pass-through semaphore for efficient concurrency control, file cell for scalable in-memory data structures and `mlog` for concurrent space allocation. Through evaluation, we show that MAX outperforms modern Linux file systems with the scaling of cores. The source code of MAX is available at github.com/thustorage/max.

8 Acknowledgement

We sincerely thank our shepherd Ric Wheeler and the anonymous reviewers for their valuable feedback. This work is supported by the National Key Research & Development Program of China (Grant No. 2018YFB1003301), and the National Natural Science Foundation of China (Grant No. 62022051, 61832011, 61772300).

References

- [1] Exim. <https://www.exim.org/>.
- [2] Fdisk. <https://en.wikipedia.org/wiki/Fdisk>.
- [3] Kvm disk cache modes. <https://documentation.suse.com/sles/11-SP4/html/SLES-kvm4zseries/cha-qemu-cachemodes.html>.
- [4] Nvm express specification. <https://nvmexpress.org/developers/nvme-specification/>.
- [5] Sqlite. <https://www.sqlite.org/index.html>.
- [6] Zoned namespaces (zns) ssds. <https://zonedstorage.io/introduction/zns/>.
- [7] Dbench. <https://dbench.samba.org/>, 2008.
- [8] Linux perf. https://perf.wiki.kernel.org/index.php/Main_Page, 2015.
- [9] NVMe. <https://nvmexpress.org/white-papers/>, 2018.
- [10] 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. SOSP '13, page 325–340, New York, NY, USA, 2013. Association for Computing Machinery.
- [11] 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 the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 69–86, New York, NY, USA, 2017. ACM.
- [12] Matias Bjørling. From open-channel ssds to zoned namespaces. https://www.usenix.org/sites/default/files/conference/protected-files/nsdi19_slides_bjorling.pdf.
- [13] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: Introducing multi-queue ssd access on multi-core systems. 07 2013.
- [14] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.
- [15] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.
- [16] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '13, pages 157–166, New York, NY, USA, 2013. ACM.
- [17] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 1–17, New York, NY, USA, 2013. ACM.
- [18] Jonathan Corbet. Xfs: the filesystem of the future? <https://lwn.net/Articles/476263/>, 2012.
- [19] Dave Dice and Alex Kogan. Bravo—biased locking for reader-writer locks. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 315–328, Renton, WA, July 2019. USENIX Association.
- [20] Dave Dice and Alex Kogan. Compact numa-aware locks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 12:1–12:15, New York, NY, USA, 2019. ACM.
- [21] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. *ACM Trans. Parallel Comput.*, 1(2):13:1–13:42, February 2015.
- [22] FaceBook. Rocksdb. <https://github.com/facebook/rocksdb>.
- [23] Intel. Breakthrough performance for demanding storage workloads. <https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-905p-product-brief.pdf>.
- [24] Intel. Intel ssd p4618 serial. <https://ark.intel.com/content/www/us/en/ark/products/series/192575/intel-ssd-dc-p4618-series.html>.
- [25] Intel. Intel® ssd 760p series. <https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/>

[consumer-ssds/7-series/ssd-760p-series/760p-series-1-024tb-m-2-80mm-3d2.html](https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/7-series/ssd-760p-series/760p-series-1-024tb-m-2-80mm-3d2.html).

- [26] Intel. Intel® ssd d7-p5600 series. <https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/d7-series/d7-p5600-series/d7-p5600-3-2tb-2-5in-3d3.html>.
- [27] Intel. Intel® ssd dc p3700 series. <https://ark.intel.com/content/www/us/en/ark/products/series/79628/intel-ssd-dc-p3700-series.html>.
- [28] Intel. Intel® ssd dc p4510 series. <https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/d7-series/dc-p4510-series/dc-p4510-15-3tb-e1-1-18mm.html>.
- [29] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. Spanfs: A scalable file system on fast storage devices. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 249–261, Berkeley, CA, USA, 2015. USENIX Association.
- [30] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. Scalable and practical locking with shuffling. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '19, 2019.
- [31] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalable numa-aware blocking synchronization primitives. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 603–615, Santa Clara, CA, 2017. USENIX Association.
- [32] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *USENIX*, 1995.
- [33] Linux Kernel. Percpu counter. https://elixir.bootlin.com/linux/v4.19.11/source/include/linux/percpu_counter.h.
- [34] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. Mvrlu: Scaling read-log-update with multi-versioning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 779–792, New York, NY, USA, 2019. ACM.
- [35] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 40(3):102–107, July 2006.
- [36] 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*, SOSP '17, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery.
- [37] Changman Lee, Dongho Sim, Joo-Young 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'15, pages 273–286, Berkeley, CA, USA, 2015. USENIX Association.
- [38] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: The design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.
- [39] Yossi Lev, Victor Luchangco, and Marek Olszewski. Scalable reader-writer locks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 101–110, New York, NY, USA, 2009. ACM.
- [40] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Write dependency disentanglement with HORAE. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 549–565. USENIX Association, November 2020.
- [41] Linux. Linux percpu_rwlock. <http://lxr.freeelectrons.com/source/include/linux/percpurwsem.h>, 2012.
- [42] Ran Liu, Heng Zhang, and Haibo Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 219–230, Berkeley, CA, USA, 2014. USENIX Association.
- [43] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 81–96, Berkeley, CA, USA, 2014. USENIX Association.

- [44] Avantika Mathur, Mingming Cao, Suparna Bhat-tacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.
- [45] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: A lightweight syn-chronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Sys-tems Principles, SOSP '15*, pages 168–183, New York, NY, USA, 2015. ACM.
- [46] Richard McDougall and Jim Mauro. Filebench. <http://www.nfsv4bat.org/Documents/nasconf/2004/filebench.pdf>, 2005.
- [47] Paul E. McKenney. Kernel korner: Using rcu in the linux 2.5 kernel. *Linux J.*, 2003(114):11–, October 2003.
- [48] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. Understanding many-core scalability of file systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 71–85, Berkeley, CA, USA, 2016. USENIX Association.
- [49] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. Sfs: Random write con-sidered harmful in solid state drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 12, USA, 2012. USENIX Association.
- [50] Daejun Park and Dongkun Shin. ijournaling: Fine-grained journaling for improving the latency of fsync system call. In *Proceedings of the 2017 USENIX Confer-ence on Usenix Annual Technical Conference*, USENIX ATC '17, pages 787–798, Berkeley, CA, USA, 2017. USENIX Association.
- [51] Christoph Rohland. tmpfs. <https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt>, 2010.
- [52] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.
- [53] Samsung. Samsung 980pro ssd. <https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/980pro/>.
- [54] Samsung. Samsung 983 dct datacenter ssd. <https://www.samsung.com/semiconductor/minisite/ssd/product/data-center/983dct/>.
- [55] Samsung. Ultra-low latency with samsung z-nand ssd. https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf.
- [56] Yongseok Son, Sunggon Kim, Heon Young Yeom, and Hyuck Han. High-performance transaction processing in journaling file systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 227–240, Berkeley, CA, USA, 2018. USENIX Association.
- [57] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main mem-ories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 323–338, Berkeley, CA, USA, 2016. USENIX Association.
- [58] ZDNet. Zoned flash: The next big thing in en-terprise ssds. <https://www.zdnet.com/article/zoned-flash-is-coming/>.

Z-Journal: Scalable Per-Core Journaling

Jongseok Kim[†], Cassiano Campes[†], Joo-Young Hwang[‡], Jinkyu Jeong[†] and Euseong Seo[†]
[†]*Sungkyunkwan University, Republic of Korea.*
[‡]*Samsung Electronics Co., Ltd.*

Abstract

File system journaling critically limits the scalability of a file system because all simultaneous write operations coming from multiple cores must be serialized to be written to the journal area. Although a few scalable journaling approaches have been proposed, they required the radical redesign of file systems, or tackled only a part of the scalability bottlenecks. Per-core journaling, in which a core has its own journal stack, can clearly provide scalability. However, it requires a journal coherence mechanism because two or more cores can write to a shared file system block, so write order on the shared block must be preserved across multiple journals. In this paper, we propose a novel scalable per-core journal design. The proposed design allows a core to commit independently to other cores. The journal transactions involved in shared blocks are linked together through order-preserving transaction chaining to form a transaction order graph. The ordering constraints later will be imposed during the checkpoint process. Because the proposed design is self-contained in the journal layer and does not rely on the file system, its implementation, Z-Journal, can easily replace JBD2, the generic journal layer. Our evaluation with FxMark, SysBench and Filebench running on the ext4 file system in an 80-core server showed that it outperformed the current JBD2 by up to approx. 4000 %.

1 Introduction

The number of concurrent cores accessing a file system is ever increasing as the number of cores installed in a system increases. However, existing file systems show poor scalability for a few file system operations [16]. Especially, because write requests coming from all cores must be serialized to be written on the single journal, the journal layer acts as a representative scalability bottleneck [8, 16, 17, 19, 20].

The serialization occurs at two points in journaling; writes to the in-memory journal data structures, and to the on-disk journal area. The in-memory data structure accesses are parallelizable to some degree by applying lockless parallel data

structures [19]. However, the parallelized memory operations, in the end, should be serialized for the on-disk journal writes. The serialization at the on-disk journal works as the more serious inhibitor for the file system scalability because the storage performance is still significantly slower than the main memory. In addition, the serialized journal writes hinder the performance gain earned from the ever-increasing internal-parallelism of modern SSDs [4].

A few research results have been proposed to achieve scalability in journaling. However, they require an explicit separation of the file system space [8], byte-addressible non-volatile memory (NVM) as the journaling device [20], or tight coupling of file system and journal layer [2]. Therefore, in order to apply them to existing systems, application modification, adoption of NVM, or radical changes to the file system design are necessary, respectively.

Not only performance but also stability and reliability are important criteria for choosing a file system. The file systems being used in production systems have obtained their reliability and performance through decades of improvement and refinement. Consequently, it is a difficult choice to migrate to a radically redesigned file system. This leads to the large demand for the scalable generic journal layer that can replace the existing ones, such as Journaling Block Device 2 (JBD2) [22].

The most intuitive approach to realize a scalable journal is having independent journal space and journal stack per-core. If the thread running on a core can write to the journal dedicated to the core independently to the other threads, the file system can achieve the complete performance scalability to the point of the maximum disk performance.

However, when two or more threads simultaneously perform write operations to the same file system block, inconsistency between write order to the in-memory buffers and that to the on-storage journals may occur. For example, as shown in Figure 1, let us suppose that core 0¹ modifies block 0 and block 1. In turn, core 1 updates block 0 and block 2.

¹For brevity, we will use *core* to denote the thread running on the core unless otherwise stated.

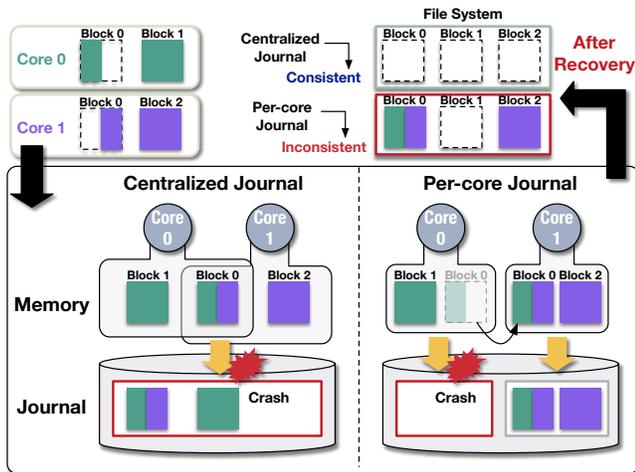


Figure 1: Inconsistency between write order to in-memory data and that to on-storage journal caused by per-core journaling without a coherence mechanism.

In this case, the journal commit issued by core 1 may finish earlier than that by core 0 when they independently operate. If a system crash occurs when core 1 committed, but core 0 did not, block 0 will be restored to a state that includes the modifications from both core 0 and core 1 during the recovery procedure. Because core 0 has not committed its modifications, the valid shape of block 0 after recovery must exclude the modifications by core 0, or have no modifications at all. This kind of inconsistency cannot happen in the conventional centralized journal.

Commonly, cores share metadata or files. Two cores have to share the metadata blocks even when they do not share any files if their files happen to be stored in the same block group [1], which is false sharing in this case. Regardless of whether being false or true, sharing blocks among cores is inevitable when the multiple cores access the same file system. Therefore, a scalable per-core journaling scheme must have a journal coherence mechanism that keeps the write order for the shared block modification.

This paper proposes *Z-Journal*, a scalable per-core journal scheme. *Z-Journal* includes a novel coherence mechanism. *Z-Journal*'s coherence mechanism allows each core to commit transactions to its journal area independently to other cores. However, when shared-block writes exist in the journal transactions, *Z-Journal* forms write-order graphs among transactions sharing blocks through *order-preserving dependent transaction chaining* and commits them with the transactions. Imposing order-constraints over transactions will be performed when checkpointing the committed transactions. Through this journal coherence mechanism, *Z-Journal* enables scalable per-core journaling while keeping crash consistency.

Z-Journal is designed to provide an identical interface to

JBD2. Therefore, it can be easily applied to the existing file systems that use JBD2, such as ext4 and OCFS2 [6], as their journal mechanism. However, to maximize the effectiveness of per-core journaling, it is desirable to eliminate false sharing among files coincidentally placed in the same block group. For this, we additionally propose a core-aware block-group allocation algorithm for the ext4 file system.

We implemented *Z-Journal* in the Linux kernel and applied it to the ext4 file system as its journal layer. For evaluation, we measured the performance and scalability of the *Z-Journal* and ext4 combination while executing *FxMark* [16], *Sys-Bench* [12] and *Filebench* [21, 23] in an 80-core server.

The remainder of this paper is organized as follows. Section 2 introduces the background and motivation of this research. Section 3 proposes the design and implementation of *Z-Journal*, and Section 4 evaluates the proposed scheme using various benchmark workloads. After the related work is introduced in Section 5, Section 6 concludes our research.

2 Background and Motivation

2.1 Design of JBD2

A file system operation usually relates to the modification of multiple file system blocks. The conventional storage devices are unable to guarantee atomic writes of multiple blocks. When a sudden crash occurs during a file system operation, only a part of the modifications may be reflected on the storage, and the file system metadata and actual data may eventually mismatch each other. The partial update of the metadata or the mismatch between metadata and data can destroy the validity of the file system.

The journaling mechanism is a measure to ensure consistency by logging the file system changes caused by an operation in the predetermined location. After the file system commits the series of changes caused by a file system operation, the journal reflects the logged changes to the file system through the checkpoint operation. Once a file system operation is committed to the journal, the journal guarantees that the changes are reflected in the file system because it can replay the committed changes even after a system failure.

JBD2 is a generic journaling layer used in the Linux kernel. JBD2 groups a series of file system changes during an interval together into a unit called a transaction. A transaction is committed to the journal area, periodically, or on the conditions explained later. When a transaction is successfully committed to the journal, JBD2 leaves the commit block at the end of the journal record. When JBD2 performs recovery after crash or failure, it checkpoints the transactions having a commit block and discards the transactions without a commit block.

A transaction undergoes a few phases during its life cycle from its creation to checkpoint. Figure 2 shows the organization of transactions in different phases. A transaction is in one of the four states: ① *running*, ② *locked*, ③ *committing*,

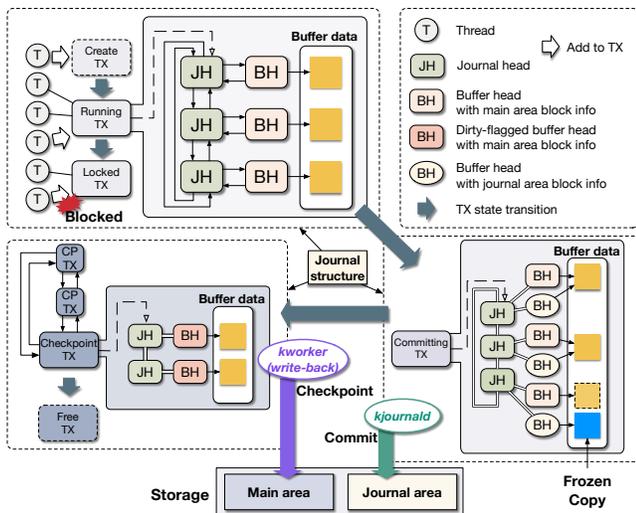


Figure 2: Organization and life cycle of a transaction in JBD2.

and ④ *checkpoint*, respectively. (The intermediate states are omitted here for brevity.) The journal structure is the central data structure of JBD2, and has three pointers pointing to the running transaction, committing transaction and checkpoint transaction list, respectively. There can be up to a single running transaction and up to a single committing transaction at a time point. The checkpoint transaction list pointer points to the head node of the doubly linked list of transactions to be checkpointed. Its head node is the oldest transaction in the list.

The buffered image of a file system block in the main memory is called a *buffer*. A buffer in the main memory is unique for a file system block. Therefore, all cores share and access the same buffer when accessing the same file system block. A buffer has its buffer head. The buffer head contains the information about the buffer and its corresponding file system block, such as the block device, logical block number, data size, etc. A buffer head is inserted in a transaction through a journal head data structure. A journal head is allocated for and bidirectionally connected to a buffer head. In a transaction, the journal heads of modified buffers are chained in a doubly-linked circular list, as shown in the upper left side of Figure 2. The oldest journal head in a transaction becomes its list head. Modifications of buffers grouped in a transaction are considered an atomic operation.

Since JBD2 only allows a single transaction to run at a time, it is trivial to order the writeback of the updated buffers. It simply writes them back they were inserted into the list of committed transactions. Later, we will explain how Z-Journal relaxes these requirements and allows multiple transactions to run concurrently.

The running transaction accommodates the modified buffers produced by file system operations. If there is no

running transaction when an operation is issued, a new transaction is created and becomes the running transaction. When a core writes to a buffer, its buffer head will attach to a new journal head, and the journal head will be inserted into the journal space of the running transaction. At this moment, the journal space must be reserved for the inserted buffers so that later the running transaction can be committed without space allocation. If the journal space is insufficient, the user-level thread performs checkpoint to free up the journal space.

Later, *kjournald*, which is a kernel thread in charge of the commit operation, starts to commit the running transaction upon one of these three conditions: (1) the transaction timeout occurs, which is by default 5 seconds; (2) the transaction capacity, which is by default a quarter of the journal area, is exhausted; or (3) the *fsync* system call is invoked by a process. Once the commit operation begins *kjournald* turns the running transaction to the locked transaction.

When the transaction enters the locked state, it cannot accommodate any more updated buffers, except the ones from the file system operation accepted to join the transaction but did not finish yet. When the last modified buffer is inserted into the locked transaction, its state is changed to the committing state. A new running transaction cannot be created while the locked transaction is waiting for its closure. Therefore, a thread issuing a new file system operation should be blocked until a new running transaction becomes available after the locked transaction becomes the committing transaction.

As shown in the bottom right corner of Figure 2, *kjournald* creates another buffer head for each buffer. This buffer head contains the information of the journal block to which the corresponding buffer will be committed. After this, *kjournald* writes the buffers of the committing transaction to their assigned journal blocks.

Usually, the original buffers remain attached to the buffer heads in the committing state. However, when a thread tries to modify the buffer included in the committing transaction before *kjournald* begins writing to the journal, the thread makes a replica of the buffer called a *frozen copy*, and replaces the original buffer with the frozen copy. The original buffer can then be freed from the committing process, and the thread can modify the buffer.

When the commit is finished, the buffer heads will be marked as dirty to denote that their buffers are required to be written to their originated file system blocks during checkpointing. After this, *kjournald* finally converts the committing transaction to a checkpoint transaction and insert it at the end of the checkpoint transaction list.

The checkpoint operation is handled by the write-back kernel thread, *kworker*, every 5 seconds, or by a user-level thread performing file operations when it finds out that there is not enough space left in the journal area for the write operation. *kworker* moves the dirty buffers of the transactions that have stayed in the checkpoint list for longer than 30 seconds to

their originated file system blocks and mark them as clean.

Because a journal head uses a separate pointer to be connected to a checkpoint transaction, a buffer can belong to a running transaction and a checkpoint transaction at the same time. When a thread tries to modify a buffer already in a checkpoint transaction, it will be inserted to the running transaction, and at the same time, its dirty flag will be cleared so that it will not be checkpointed. In this situation, the running transaction is allowed to modify the buffer.

kjournald later frees the clean buffers from the checkpoint transaction. It also frees the empty checkpoint transactions from the checkpoint list and the corresponding commit transactions from the journal area to make free space.

In case of a system crash, kjournald initiates the recovery process. It searches for the committed transactions in the journal area, and replays them in order. This guarantees that the file system remains consistent and committed data are preserved in the file system.

2.2 Scalability Bottlenecks in JBD2

The current JBD2 design has multiple scalability bottleneck points as follows.

At a given time point, there is only a single running transaction, which is the only transaction that can accept the modified buffers. Therefore, when multiple cores perform file operations in parallel, they have to compete for the lock acquisition for the running transaction [19].

Secondly, when the running transaction is closed, a new running transaction can be created only after the current committing transaction finishes, and the closed transaction becomes the committing transaction. Therefore, when the locked transaction waits for the committing transaction to finish, all cores that issue file operations must wait altogether. The larger the number of waiting cores, the more this convoy effect adversely impacts the overall file system throughput.

Last but not least, the current JBD2 does not fully utilize the internal-parallelism provided by the modern NVMe SSDs because the kjournald thread solely issues a serialized stream of buffer writes to the storage. To utilize the high-performance of modern storage devices, the journal mechanism should be able to commit in parallel.

These problems commonly stem from that there is only a single running transaction and a single committing transaction in the system. However, blindly parallelizing the running and committing transactions or entire journal stack for achieving scalability complicates keeping the write orders of the shared blocks as stated in Section 1 because a buffer head can belong to only a single transaction in the current design. If a buffer head is allowed to simultaneously exist in multiple transactions, the coherence mechanism to guarantee the write order consistency across multiple transactions to the shared buffers is necessary. This problem was referred to as a *multi-transaction page conflict* by Won et al. [24].

An approach that makes a frozen copy of a buffer whenever the file system modifies it and inserts the copy to the journal transaction instead may allow simultaneous writes to the same buffer coming from multiple threads. However, it still requires the ordering of block copies when committing a transaction so that the preceding block modifications are guaranteed to be committed before. This will be another serialization point. In addition, adding buffer copy operations to the file system write path will notably retard the write latency. Therefore, journaling with a parallel transaction requires an efficient coherence mechanism that can preserve the write orders to the buffers, while allowing as much parallel buffer modification and independent transaction management as possible.

3 Our Approach: Z-Journal

In Z-Journal, each core has its journal area on the storage device, and its kjournald, which handles the commit operation. Because kjournald is bound to each core and executed locally, this also improves the memory access locality in non-uniform memory access (NUMA) systems. The journal stack of each core has the running transaction, committing transaction, and the checkpoint transaction list the same as JBD2, and their life-cycles are identical as well.

This per-core journal approach removes the aforementioned serialization points in the journal layer and thus obtains scalability. First, because each core has its running transaction, a thread does not need to compete with the other threads for acquiring access to the running transaction. Second, when the running transaction of a core closes, and the core waits for the previous committing transaction to finish to create a new running transaction, this waiting only applies to that core. Therefore, this convoy effect is confined in the core boundary. Finally, because multiple kjournald are able to commit in parallel, Z-Journal can fully utilize modern high-performance storage devices.

However, as stated earlier, the per-core journal approach must deal with the mismatch between the in-memory buffer write order and the on-disk transaction commit order to guarantee the crash consistency.

3.1 Analysis of Journal Coherence Problem

In the per-core journal design, a thread inserts modified buffers to the running transaction allocated for the core it runs on. However, the buffer may have already belonged to a transaction of other cores as shown in Figure 1. Figure 3 categorizes this situation into three cases. In Figure 3, two cores modify two unrelated buffers, but also both modify a shared buffer. The same as JBD2, when a core tries to write to a buffer that is in a checkpoint transaction, the buffer can be inserted to the running transaction of the core without breaking the crash consistency because its last image is safely stored

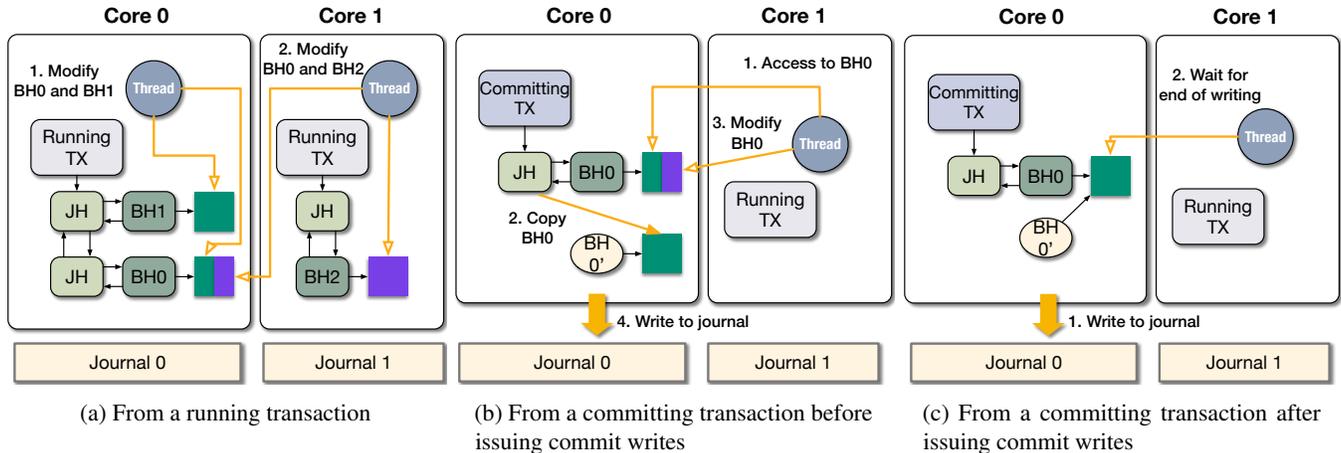


Figure 3: Three different cases that a running transaction tries to access a buffer head that already belongs to another transaction in the per-core journal system.

in the journal area. Therefore, this case is not considered in the journal coherence mechanism.

In Figure 3a, both cores modify the shared buffer before either begins to commit. Since we only maintain one copy of the shared buffer, we can no longer separate the updates made by the two cores. However, if we treat both transactions of these two cores as parts of one large transaction, we can allow uninterrupted parallel access to the shared buffers while guaranteeing the crash consistency. Their commits should be considered as valid only when both are committed; otherwise, both must be voided.

In Figure 3b and Figure 3c, one core begins to commit its running transaction before the other touches the shared buffer. In Figure 3b, the buffer has not yet been scheduled for commit, so we can create a frozen copy. In JBD2, the frozen copy will be attached to the buffer head for the commit to the journal area, and the buffer head for the original buffer is kept in the committing transaction. Because the original buffer head is no longer used by the committing transaction, it can be modified by the other cores. This allows an immediate update of the shared buffer in Figure 3b. In Figure 3c, the shared buffer is already being written to the journal area by the first core. At this point, it is too late to make a copy of the buffer for concurrent modifications, since the file system page cache would continue to point to the copy that is being written. Therefore, core 1 must wait for core 0 to complete its commit operation before modifying the shared buffer. If a committing transaction always creates frozen copies for all of its buffers and use them for commit block writes, this waiting can be eliminated.

However, in both Figure 3b and Figure 3c, to guarantee crash consistency, core 1's commit must be performed after core 0's commit finishes. This serialization may suspend core 1's file system operations because a new running transaction can be created only after core 1's current running transaction

turns into the committing transaction, which again can be possible only after the current committing transaction finishes. If there is a long chain of transaction dependency, this will result in poor scalability. However, if we can enforce a rule that a committed transaction can be checkpointed only after the transactions it depends on are committed, we can allow both of core 0's committing transaction and core 1's running transaction to be committed independently.

Analysis of the three conflicting cases revealed that the conditional recognition of the committed transactions, which checkpoints the committed transactions only when their dependent transactions are committed as well, allows them to be committed independently to each other. In addition, the proactive use of frozen copies for committing allows immediate writes to the buffers being shared with the committing transaction. Based on this observation, we propose the journal coherence mechanism for Z-Journal.

3.2 Journal Coherence Commit

The journal coherence mechanism of Z-Journal imposes the write order between transactions during checkpoint so that each core can commit its transactions without being interrupted by activities of other cores. In Z-Journal, the committed transaction will be considered as *valid* only after all transactions preceding the transaction in the write order are committed. Z-Journal checkpoints only the valid commits.

To realize this, Z-Journal should be able to identify the ordering relationships between transactions, and to record them in the transaction commit. We propose order-preserving transaction chaining for this.

In Z-Journal, a transaction maintains the information about the transactions having ordering relationship with it by recording their unique identifiers into its *chained-transaction lists* as shown in Figure 4. The number of lists in a transaction is

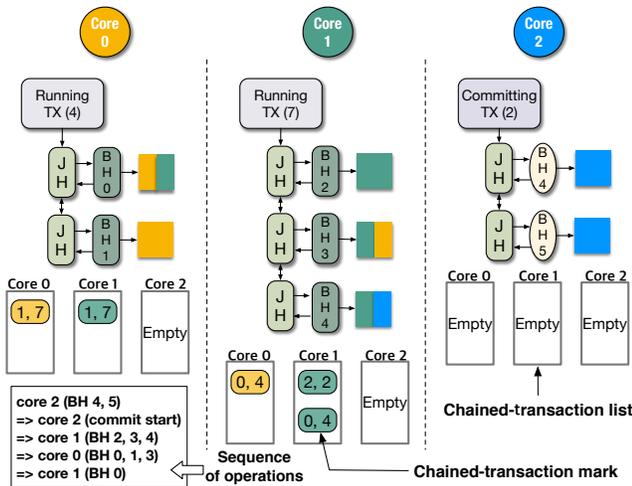


Figure 4: Example of order-preserving transaction chaining.

equal to the number of cores in a system; hence each core can update its corresponding list in any transactions without acquiring a lock. A transaction has a transaction identifier (ID), which monotonically increases in a per-core journal. Accordingly, a unique identifier of a transaction is a pair of a core ID and a transaction ID. When a transaction has an entry of (j, t) in its chained-transaction lists, the transaction is valid only after transaction t of core j is committed.

When two running transactions share a buffer, as discussed on Figure 3a, they should form a bi-directional transaction chain. If a running transaction tries to write to a buffer that belongs to a committing transaction as in Figure 3b, they should form a uni-directional chain that the running transaction follows the committing transaction.

For example, in Figure 4, buffer 4, written by transaction 2 of core 2, is about to be modified by transaction 7 of core 1. Because transaction 2 of core 2 is already in the committing state, core 1 leaves (2,2), which means transaction 2 of core 2, in core 1's list of transaction 7. By this, an uni-directional chain is formed between (1,7) and (2,2).

When core 0 writes to buffer 3, core 0 is allowed to do so while buffer 3 remains in journal 1's running transaction. In such case, these two running transactions must be considered as a single super-transaction. Therefore, core 0 leaves (0,4) in core 0's list of journal 1's running transaction and (1,7) in core 0's list of journal 0's running transaction at the same time. This forms an all-or-nothing relationship on the two running transactions. Later, when core 1 writes to buffer 0, core 1 will leave (1,7) in core 1's list of journal 0's running transaction and (0,4) in that of journal 1's because core 1 is not aware of the chained-transaction marks left by core 0 at this moment.

When each core independently modifies non-shared files, no chain will be created over their transactions. In this situa-

tion, the order in which the transactions are committed may be different from the original write operation order. For example, a process can perform metadata operations to different files on two different cores, respectively, and the second may commit when the first does not. The POSIX semantics does not guarantee the durability of write operations before finishing `fsync` of the corresponding file descriptor. Therefore, reversing the commit order between the transactions that have no ordering relationship does not violate the POSIX semantics. Even when the first invokes `fsync` before the second commits, the second may commit before the first and this is also allowed in the POSIX semantics because `fsync` is supposed to commit write operations only of a given file descriptor.

When a synchronous write from `O_SYNC` or `O_DSYNC`, is issued, the transaction chains related to the current write, if existing, should have been already formed or will be formed by the current write. Therefore, after every write operation, Z-Journal commits only the running transactions chained to the running transaction of the current write. However, when a core calls `fsync`, Z-Journal enforces all cores to commit their running transactions because `fsync` is supposed to flush all transactions related to the given file descriptor, and they can be in any core without being chained to the running transaction of the current core.

Since `fsync` does not add a new buffer head to the transaction nor allocate journal space, the commit time takes up most of the `fsync` latency. In Z-Journal, the commit operation must be performed in all cores to finish `fsync`, but the delay from it is not significant because the commit operation is executed in parallel in each core. Rather, when `fsync` is called in parallel on multiple cores, it is significantly advantageous in terms of throughput because multiple `fsync` invocations, which must be serialized in JBD2, can be parallelized in Z-Journal.

When a transaction enters the committing state, Z-Journal proactively creates frozen copies of its buffers regardless of their sharing states to prevent buffer update from waiting for finishing the commit operation as shown in Figure 3c. Through this *proactive frozen copy* approach, a committing transaction is disconnected from the original buffers and accesses only their frozen copies. Therefore, in Figure 4, when core 1 writes to buffer 4, buffer 4 can be inserted to transaction 7 without waiting because transaction 2 of journal 2 is using the frozen copy of buffer 4.

The proposed order-preserving transaction chaining scheme enables Z-Journal to keep track of write orders among transactions in a scalable and efficient way. Based on this, Z-Journal puts off the enforcement of write-order constraints to the checkpoint time and allows cores to independently commit transactions regardless of their sharing status. Because the usual checkpoint interval is a lot longer than the transaction life span, it is highly likely that almost all transactions become valid at the time of the checkpoint. Therefore, Z-Journal's journal coherence mechanism to the checkpoint duration is expected to be minimal.

The proactive frozen copy approach enables the immediate sharing of a buffer that is currently in use of a committing transaction. Combined with the order-preserving transaction chaining, this enables all transactions to simultaneously proceed to the checkpoint state without waiting. However, the proactive frozen copy generates a significant amount of memory copy operations and increases memory consumption. In addition, the order-preserving transaction chaining requires additional writes to the lists, although they are lockless. Nevertheless, because these overheads involve the in-memory structures, not the on-disk journal structures, their impact on the scalability and overall performance will be negligible compared to the expected benefits.

3.3 Journal Coherence Checkpoint

The same as JBD2, in Z-Journal, there is a single kworker thread in the system, and it periodically performs the checkpoint operation. Also, the same as JBD2, a user-level thread can conduct checkpoint when its write operation is delayed due to the insufficient free journal space.

As stated earlier, not all committed transactions become objects of checkpointing in Z-Journal. To implement this, when kjournald changes the state of a transaction to the checkpoint state, it skips over setting dirty flags of transaction's buffers. Instead, when kjournald converts its running transaction to the committing transaction, it checks whether the transactions in its checkpoint transaction lists are valid. If a transaction turns out to be valid, its buffers will be marked as dirty. Later, they will be checkpointed by kworker, which periodically iterates and checkpoints dirty buffers in the background.

For a committed transaction to be valid, its direct preceding transactions must be not only committed but also valid. Therefore, to check the validity of a committed transaction, kjournald traverses the transaction chain graph, which was created from the chained-transaction marks of the transaction and its ancestors, and checks whether all of their ancestors are valid. Every transaction has a field that shows its validity. If every ancestor of a transaction is identified as valid during the search, kjournald sets its validity field to prevent redundant search over its ancestors in the future.

This validity check is performed not only by kjournald, but also by user-level threads. When kjournald finds out an uncommitted ancestor, it stops the search and starts processing the next checkpoint transaction. However, in such a case, a user-level thread will initiate committing the uncommitted ancestor. It continues after the commit finishes because it cannot proceed with its file system operation without freeing journal space.

A valid transaction can be checkpointed anytime independently from its chained transactions. Therefore, the checkpoint order of valid transactions may be different from their dependent transaction orders. However, the removal of a transaction from a journal can be allowed only after its chained

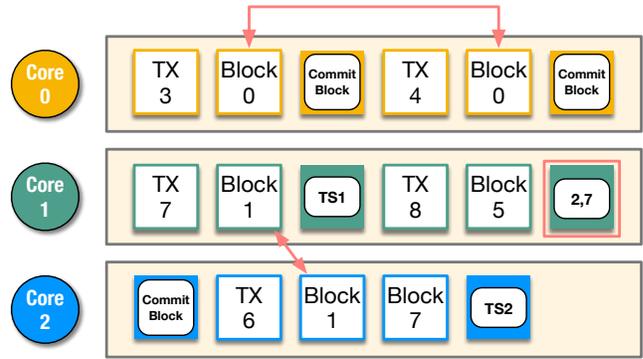


Figure 5: Snapshot of per-core journals after system crash.

transactions are all checkpointed because all chained transactions must be replayed together during the recovery.

The commit operation is conducted using frozen copies. However, the journal heads of a checkpoint transaction point to their original buffers. Therefore, checkpointing a valid transaction always updates the file system blocks with the up-to-date images of dirty buffers regardless of the checkpoint order.

3.4 Recovery Procedure

A committed transaction stored in a journal area has three kinds of blocks the same as JBD2: transaction descriptor blocks, data blocks, and a commit block. The transaction descriptor blocks, which store the information about the following data blocks, are located at the head of a committed transaction. Next, the data blocks are stored. Finally, The commit block is written to indicate the successful completion of the commit.

In Z-Journal, the commit block also stores the chained-transaction lists of the transaction. In addition, the commit block also has the timestamp [9] of the commit start time for the global ordering of transaction commits across multiple journals.

The recovery process first searches for the transactions with the commit block from all journals to find the valid transactions. After this, it creates the transaction order graphs based on their chained-transaction lists. Similarly to the checkpoint procedure, the recovery process traverses the graphs to find valid transactions. Finally, it updates the file system blocks with the latest buffer images from the valid transactions.

When a buffer redundantly appears in two valid transactions connected through an ordered chain, the buffer image from the latter transaction in the graph is the latest one, which will be used for recovery. If two transactions are tied together through a bi-directional chain constructed from sharing between two running transactions, a buffer cannot exist in both transactions simultaneously. Finally, when two transactions

are not chained together but have the same buffer, the buffer image of the latter transaction, which is determined by the timestamp, will be chosen because this case means that the latter transaction overwrote the buffer after the former transaction completely committed. These rules are transitively applied to the cases involving multiple transactions.

Figure 5 shows the snapshot of the per-core journals after a system crash. Transaction 8 of core 1 is invalid, although it has the commit block because the transaction 7 of core 2, which must precede it, is not written in the journal. Therefore, it is discarded in the recovery process. Block 0 is in transaction 3 and 4 of core 0, both of which are valid transactions. However, transaction 4 has the larger transaction ID, block 0 of transaction 4 will be restored. Block 1 also appears in transaction 7 of core 1 and transaction 6 of core 2 at the same time. They do not have a dependency relationship. Therefore, the recovery process compares the timestamps of both transactions to determine the latest buffer image to restore, which is that of transaction 6 of core 2 in this case.

3.5 Core-Aware Block Group Allocation

Block grouping is leverage inherited from the legacy spinning disks to provide a faster seek time [5]. The block group allocator of ext4 decides which block group a new inode or data block should be allocated in. The current block group allocator aims at increasing the access locality and minimizing seek times to obtain performance benefit from the underlying spinning disks [13].

The block group allocator of ext4 disperses the directory allocation over as many block groups as possible. However, when creating a file, it tries to place the inode of the new file in the same or nearby block group with its parent directory. It allocates file’s data blocks in the same block group with the file inode when the file size is smaller than a predefined value, `stream_req`. When larger than that, data blocks will be allocated from the last block group in which the data blocks for a large file were allocated. If the block group cannot accommodate the request, the block group allocator will sequentially try the following block groups.

The current block group allocator does not benefit when using a flash SSD because it does not have seek time. On the contrary, it increases false sharing of metadata among cores because the allocated blocks are unevenly distributed over a few block groups, and the block group placement of files and directories are blind to who will access them.

In Z-Journal, as sharing between transactions gets more frequent, the lengths of transaction chains tend to be longer. The large transaction order graphs will incur large checkpoint overhead. Therefore, we propose a *core-aware block group allocator* for ext4 that allows the group of blocks requested by one core to be allocated exclusively to other cores as much as possible.

When i -th core requests a block or metadata entry, the pro-

		Specification
Processor	Model	Intel Xeon Gold 6138 × 4 sockets
	Number of Cores	20 × 4
	Clock Frequency	2.00 GHz
Memory		DDR4 2666 MHz 32GB × 16
Storage		Samsung SZ985 NVMe SSD 800GB
OS	Kernel	Linux 4.14.78

Table 1: System configurations for evaluation.

posed block group allocator sequentially checks from block group $\lfloor \frac{\text{number of block groups}}{\text{number of cores}} \rfloor \times i$ to find a block group that can accommodate the requested item. The data block allocation for a file is served in the same way regardless of the file size.

This simple core-aware block group allocator is proposed for analyzing the benefit from reduced false sharing, not for production use, and does not consider the long-term consequences from the core-partitioned distribution of allocated blocks and the interactions with other performance-sensitive factors. The in-depth research on core-aware or sharing-aware block group allocators is beyond the scope of this paper.

4 Evaluation

In this section, we evaluate Z-Journal to verify its performance and scalability for various file system operations under different sharing conditions. In addition, We also analyze the overhead and benefit of the proactive frozen copy scheme and Z-Journal’s `fsync` handling mechanism. Finally, we show the overall file system performance and scalability of Z-Journal for benchmarks imitating real-world workloads.

4.1 Evaluation Environment

Table 1 shows the system configurations used for the evaluation. We implemented Z-Journal in the Linux kernel² and modified the ext4 file system to recognize per-core journals and to use Z-Journal instead of JBD2. We also modified the block group allocator of ext4 as described in Section 3.5. In addition, we modified `mke2fs` to format an ext4 file system to have multiple journals. The ext4 file system was modified so that the super block can have multiple journal control structures and the mount operation recognizes them.

We compared the performance of Z-Journal (denoted as ZJ on the graphs) with ext4 with JBD2 (denoted as JBD2 on the graphs), and ext4 without journaling (denoted as *no-journal* on the graphs). Because we are not aiming at the scalability of the overall file system, the performance of ext4 without journaling can be considered the best possible value Z-Journal can achieve. We also measured the performance of Z-Journal without proactive frozen copy (denoted as *w/o PFC* on the graphs), and without core-aware block group allocator

²The source code of the Z-Journal-patched Linux kernel is available at <https://github.com/J-S-Kim/journal>

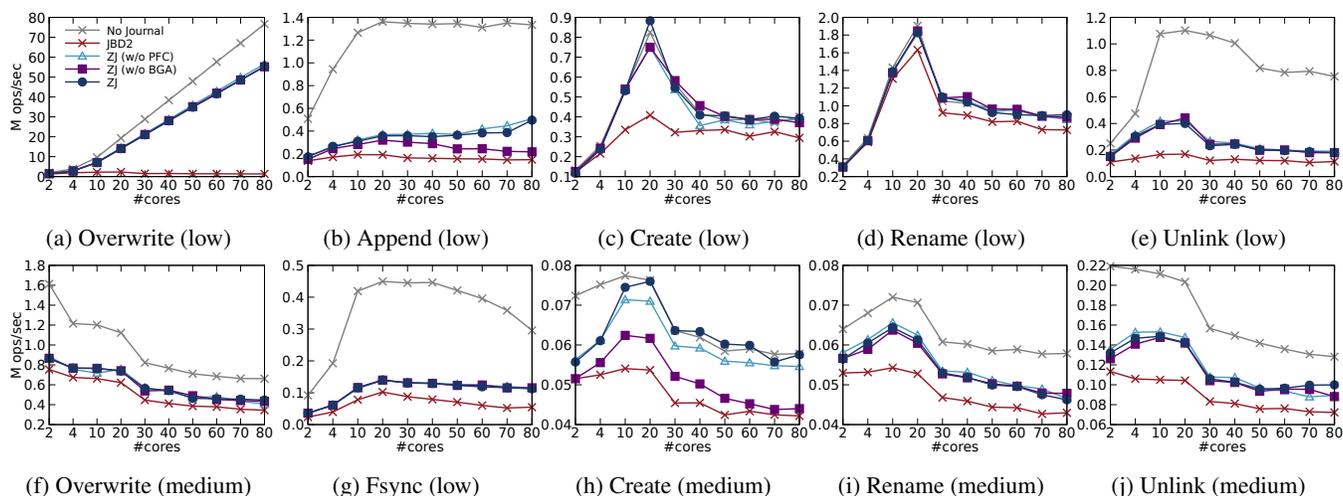


Figure 6: Throughput of FxMark write workloads while varying number of cores under different sharing conditions.

(denoted as *w/o BGA* on the graphs). All experiments were conducted in the `data=journal` mode [18] (The results obtained in the `data=ordered` mode are also presented in the appendix).

4.2 Scalability of File System Operations

To assess the scalability of Z-Journal when performing file operations under various block sharing conditions, we used FxMark [16]. The write workloads of FxMark consists of overwrite, append, fsync, create, unlink, and rename operations executed in the low and medium sharing level, respectively. In the low sharing condition, each core performs the target file system operation for private files in its dedicated directory. In the medium sharing level, the overwrite workload lets all cores access the same file, and the other workloads perform the given operation in the same directory. The append and fsync workloads provide only the low sharing level mode. Figure 6 shows the experiment results.

The favorable condition for Z-Journal against JBD2 is where the file system scalably performs, but the resulting performance is poor for the serialization at the journal layer. overwrite (low) barely modifies metadata and frequently writes to data blocks of files stored in non-shared per-core directories. Therefore, it is the most favorable workload in FxMark.

Z-Journal showed a close performance to no-journal for overwrite (low) excluding the slow down from the double writing overhead, and its performance scaled well to the number of cores. Z-Journal showed 41 times higher throughput in comparison to JBD2 at 80 cores. However, in the case of overwrite (medium), the file system scalability was poor due to sharing, and the performance of the journal layer was also poor for the same reason. Even in this case, Z-Journal gained 30% of performance improvement at 80 cores compared to JBD2 through its parallel journaling mechanism.

In the case of append (low), the difference between no

journal and journal group was very large. The metadata manipulation overhead without journaling is very low because delayed allocation is possible when allocating a new data block. Z-Journal without BGA scaled gently up to 20 cores, but performance decreased after that. This was due to increased false sharing caused by the current block group allocator. When the core-aware block group allocator was used, the performance steadily increased up to 80 cores. Z-Journal showed 3.34 times the performance of JBD2 at 80 cores.

For create (low) and rename (low), the journal’s scalability bottleneck was not revealed due to the scalability problem of the ext4 file system [16]. However, since commits were performed in parallel, there were 33 % and 24 % performance improvements at 80 cores, respectively. In terms of create (medium), Z-Journal without BGA obtained performance improvement of up to 15 % at 10 cores and about 4 % at 80 cores. Z-Journal showed similar performance to no-journal. Because it allocated block groups for each core, although the files were in the same directory, metadata sharing was greatly reduced, and this leads to improved performance. This performance gain was mostly from the file system, not from the journal layer. This result shows that the block group allocator in a file system is one of the major obstacles to scalability.

In the case of unlink, no-journal also showed scalability characteristics similar to create and rename. The big difference in performance between no-journal and the journaling group is from the heavy checkpointing operations occurring in the measurement interval caused by the preparation stage, in which the large file set was created, and their data blocks were written. For unlink (medium), Z-Journal achieved the maximum performance improvement of 42 % at 10 cores, and 38 % at 80 cores compared to JBD2.

In the fsync workload, where all cores write to their files and call `fsync`, Z-Journal also showed an average throughput improvement of 70 % in comparison to JBD2. When a core

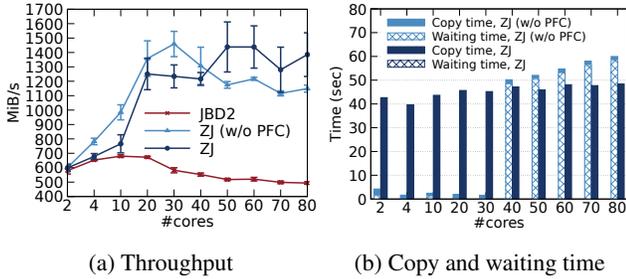


Figure 7: Throughput and journaling delay of SysBench.

called `fsync` in Z-Journal, all cores were suspended because they had to commit their running transactions together. However, the commit of each core in Z-Journal was a lot faster than that in JBD2 because per-core `kjournald` could commit in parallel while a single `kjournald` had to process alone.

Z-Journal showed a scalability pattern similar to that no-journal when the file system’s scalability was good in a low sharing situation. Even when the scalability of the file system was poor, its parallel committing scheme achieved superior performance compared to JBD2. However, in the workloads where shared writes and metadata updates frequently occurred, the performance gain was diminished due to the validity check procedure during the checkpoint process.

In the experiment for each file operation, most of the performance was better when proactive force copy was not applied. Only, low sharing create and medium sharing create achieved an average of 4 % and 5 % improvement by applying proactive force copy, respectively. It was because FxMark’s write workloads mainly produced metadata sharing and not much file sharing. If write accesses to the shared block do not occur frequently, proactive force copy will generate only meaningless overhead.

4.3 Analysis of Design Components

We used SysBench [12] to analyze the performance of Z-Journal and the impact of proactive force copy when data block write sharing frequently occurs. SysBench performs random overwrites on files in the file set created in advance. Therefore, when the number of cores increases, the occurrences of metadata block and data block sharing increase as well. We configured SysBench to randomly overwrite 4 KB data over 80 files, each of which has 1 GB size.

Figure 7a shows the throughput measured while increasing the number of cores, and Figure 7b shows the sum of the journal waiting time and the time to create frozen copies in Z-Journal and Z-Journal without proactive frozen copy, respectively. The waiting time refers to the time for a running transaction to wait for the committing transaction having the shared buffers to finish.

Both proactive frozen copy and waiting for shared buffers can be carried out by `kjournald` as well as user-level threads

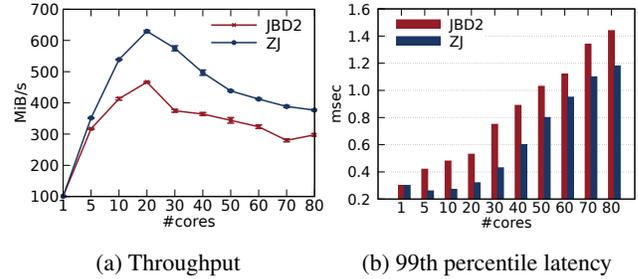


Figure 8: Throughput of `fsync`-invoking SysBench and tail latency of `fsync` operations observed.

writing to files. However, proactive frozen copy is mostly performed by `kjournald` in the background. Because it is off from the write path, the proactive frozen copy has little impact on the user-perceived performance. At 50 cores in Figure 7b, the copy time in Z-Journal and the waiting time in Z-Journal without PFC are similar, but unlike Z-Journal without PFC waiting in the write path, proactive frozen copies are mostly conducted by `kjournald`. Therefore, Z-Journal showed a 22 % better performance than Z-Journal without PFC.

The sudden increase in waiting time at 50 cores in Figure 7b was because, as mentioned in Section 3.1, long chains of dependent transactions were created. On the other hand, the copy time did not significantly increase even though the throughput of Z-Journal increased. This is because SysBench writes to the same file set repeatedly, and therefore, the number of shared buffers is limited.

As a result, as shown in Figure 7a, up to 40 cores Z-Journal without PFC outperformed Z-Journal by an average of 13 %. However, after 50 cores, proactive frozen copy improved the average throughput by 19 %. Based on this observation, we conclude that the proactive frozen copy scheme should be applied adaptively to the degrees of parallelism and cross-core file sharing.

To assess the throughput and latency of `fsync` on Z-Journal, we altered SysBench so that it calls `fsync` once every 100 write operations, and measured its throughput and 99th percentile tail latency of the `fsync` operation. Considering that `fsync` is intensively performed while cores are writing on a shared data set, this can be regarded as a notably hostile condition for Z-Journal.

As shown in Figure 8a, Z-Journal still performed better than JBD2 at all core counts. However, as the number of cores increased, unlike the results of the original SysBench, the throughput of Z-Journal also decreased similar to JBD2 because the validity check overhead of `kjournald` offset a significant part of the performance improvement earned from parallelized journaling. However, as shown in Figure 8b, the tail latency of `fsync` on Z-Journal was significantly better than that on JBD2. It was 18 % shorter at 80 cores even though the difference between JBD2 and Z-Journal narrowed as the number of cores increased as expected in Section 3.2.

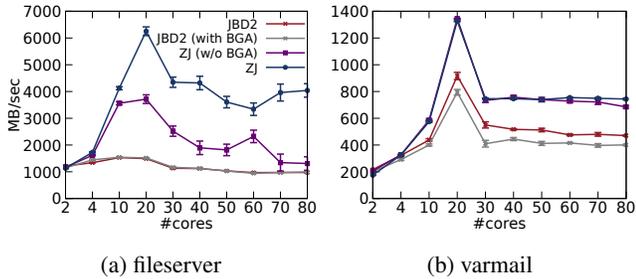


Figure 9: Throughput of Filebench workloads.

4.4 Overall File System Performance

To assess the performance impact of Z-Journal in the environment, which has a mixture of reads, writes, and metadata operations, we used the *fileserver* and *varmail* workloads of Filebench [21,23]. *fileserver* repeatedly conducts create, write, append, read, and deletion operations over a file set of relatively large files. *varmail* also continually performs create, read, append, and `fsync` operations over a file set of small files.

Figure 9a shows the throughput of *fileserver*. Since a large number of cores repeatedly created and updated files in a small number of directories, the throughput of JBD2 peaked at 10 cores and decreased as the number of cores increased. Z-Journal showed significantly better performance than JBD2 in every case. However, in the absence of BGA, false sharing increased rapidly with the increase in the number of cores, and the gap between the throughput of Z-Journal and that of JBD2 gradually shrunk as the number of cores increased. BGA significantly reduced false sharing and enabled Z-Journal to achieve 315 % performance improvement. When BGA was applied to JBD2, no throughput improvement was observed. This tells that the scalability of JBD2 was not limited only by metadata sharing. Z-Journal performed best at 20 cores, and the performance gently declined beyond that point due to the NUMA effect [7] resulting from the rapid increase in remote memory access across domains.

Figure 9b shows the throughput of *varmail*. *varmail* also showed a similar throughput change pattern with *fileserver*, and Z-Journal improved performance by 58 % at 80 cores. However, unlike *fileserver*, *varmail* frequently called `fsync`, and the performance improvement for `fsync` by Z-Journal is smaller than that for other file operations. Therefore, the performance gap between JBD2 and Z-Journal was smaller than that in *fileserver*. The effect of false sharing reduction from applying BGA was mostly concealed by the frequent suspension for processing `fsync`. As a result, BGA could obtain only 9 % performance improvement in comparison to Z-Journal without BGA. Rather surprisingly, when BGA was applied to JBD2, the performance deteriorated by up to 26 %. This was because the number of committed blocks was significantly amplified from reduced metadata sharing.

5 Related Work

Min et al. studied the scalability of file systems on manycore systems and left insights to design scalable file systems [16].

SpanFS [8] achieved scalability by partitioning files and directories into multiple domains. While this partitioning enables parallel write operations to each domain, careful data partitioning is important to achieve scalability. When two cores each write to two files in the same domain, SpanFS needs to serialize the write operations although they are independent of each other. In contrast, Z-Journal does not require explicit data partitioning and minimizes the serialization of journaling operations.

IceFS [15] also partitioned files and directories for performance isolation but its main goal was not multi-core scalability since multi-threads on a single partition (called a cube) can cause scalability bottleneck.

ScaleFS [2] decouples an in-memory file system from an on-disk file system and allows scalable file system operations since highly concurrent data structures are used in the in-memory file system. File system modifications are flushed to on-disk file system structures through per-core parallel journaling. However, the dual file system approach with the integrated journal layer of ScaleFS cannot be applied to the existing file systems. For example, journaling in ScaleFS can have multiple images of a buffer in memory by logging changes to the buffer per-core. This allows parallel journaling. However, in ext4 and most other conventional file systems, a buffer in memory must be unique and up-to-date. Z-Journal maintains this invariant while allowing parallel commits.

A few studies have addressed the scalability bottleneck caused by coarse-grained locking during write operations. They proposed to use a fine-grained range lock to increase the concurrency of write operations [10, 14]. The use of a file-grained lock is complementary to our approach.

iJournaling [17] proposed to use a per-file journal transaction for `fsync`. Accordingly, only necessary blocks need to be flushed to reduce the `fsync` latency. Since the per-file transaction uses a logical logging scheme, concurrent `fsync` processing can be possible. However, it also maintains the original file system journaling, causing serialization during write operations.

Son et al. [19] proposed to improve the concurrency of JBD2 by aggressively using concurrent and lock-less data structures and multi-threaded journal writes. While their approach enhanced the concurrency of journaling, the inherent serialization from committing to a single journal area remains.

BarrierFS [24] improved the concurrency of journaling by allowing multiple committing transactions in a journal file, which improves the concurrency of journaling. However, it has a limited concurrency due to waiting-based dependency handling, which is eliminated by Z-Journal's journal coherence mechanism.

The journaling mechanism for block devices is unsuitable

for NVM due to the write amplification of metadata journal. Chen et al. [3] proposed a fine-grained metadata journal mechanism optimized for journaling in NVM aiming at reduction in write amplification.

Sul et al. [20] proposed an NVM-optimized journaling scheme in which the use of parallel journals is similar to our proposal. However, its journaling operation aggressively exploits NVM's byte-addressable characteristic, which hinders its application to block devices.

Koo et al. [11] analyzed blocking of I/O operations due to the transaction in the locked state. To resolve this issue, they proposed a transaction lock-up elimination scheme that optimizes the transaction commit procedure.

6 Conclusion

Most modern file systems use the journal to guarantee crash consistency. However, because conventional journaling schemes are performed serially from transaction generation in memory to journal commit on disk, it acts as a serious scalability bottleneck when file system operations are run simultaneously in many cores.

In this paper, we proposed Z-Journal, a scalable journal design using per-core journals, which retains the interface of JBD2. Z-Journal includes a journal coherence mechanism, which provides complete parallelism for unshared buffer modification and guarantees crash consistency by order-preserving transaction chaining for shared buffer modification.

Our evaluation showed that Z-Journal achieves a steady increase in throughput in journal-bottlenecked workloads, showing up to approximately 4000% improvement in comparison to JBD2. It also showed 29% throughput improvement on average even under unfavorable conditions by allowing parallel journaling.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Rusty Sears, for their valuable suggestions for improving this paper.

This research was supported by Samsung Electronics, and by the Institute of Information and Communications Technology Planning and Evaluation funded by the Ministry of Science and ICT (MSIT), Korean Government, (Research on High Performance and Scalable Manycore Operating System) under Grant 2014-3-00035.

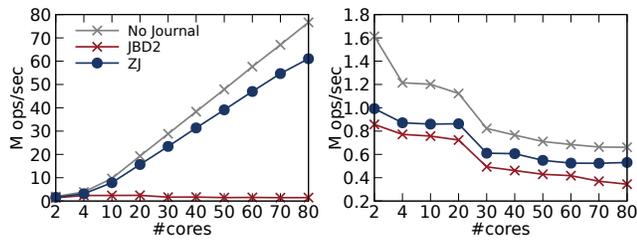
References

- [1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*, chapter 41, page 4. Arpaci-Dusseau Books, 1.00 edition, August 2018.
- [2] 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 the 26th Symposium on Operating Systems Principles (SOSP)*, pages 69–86. ACM, 2017.
- [3] Cheng Chen, Jun Yang, Qingsong Wei, Chundong Wang, and Mingdi Xue. Fine-grained metadata journaling on NVM. In *32nd Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–13. IEEE, 2016.
- [4] F. Chen, R. Lee, and X. Zhang. 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 (HPCA)*, pages 266–277, 2011.
- [5] Kevin D Fairbanks. An analysis of Ext4 for digital forensics. *Digital investigation*, 9:S118–S130, 2012.
- [6] Mark Fasheh. OCFS2: The Oracle clustered file system, version 2. In *Proceedings of the 2006 Linux Symposium*, volume 1, pages 289–302. Citeseer, 2006.
- [7] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. Challenges of memory management on modern NUMA system. *Queue*, 13(8):70–85, 2015.
- [8] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: a scalable file system on fast storage devices. In *2015 USENIX Annual Technical Conference (ATC)*, 2015.
- [9] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. A scalable ordering primitive for multicore machines. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys '18*, New York, NY, USA, 2018. Association for Computing Machinery.
- [10] June-Hyung Kim, Jangwoong Kim, Hyeongu Kang, Chang-Gyu Lee, Sungyong Park, and Youngjae Kim. pNOVA: Optimizing shared file I/O operations of NVM file system on manycore servers. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSYS)*, pages 1–7, 2019.
- [11] Kyoungho Koo, Yongjun Park, and Youjip Won. LOCKED-Free journaling: Improving the coalescing degree in EXT4 journaling. In *Proceedings of IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2020.
- [12] Alexey Kopytov. SysBench: A system performance benchmark, 2004.

- [13] Aneesh Kumar KV, Mingming Cao, Jose R Santos, and Andreas Dilger. Ext4 block and inode allocator improvements. In *Linux Symposium*, volume 1, 2008.
- [14] Chang-Gyu Lee, Hyunki Byun, Sunghyun Noh, Hyeongu Kang, and Youngjae Kim. Write optimization of log-structured flash file system for parallel I/O on manycore servers. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYS-TOR)*, pages 21–32, 2019.
- [15] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 81–96, 2014.
- [16] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (ATC)*, 2016.
- [17] Daejun Park and Dongkun Shin. iJournaling: Fine-grained journaling for improving the latency of fsync system call. In *2017 USENIX Annual Technical Conference (ATC)*, 2017.
- [18] Vijayan Prabhakaran, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *USENIX Annual Technical Conference, General Track*, volume 194, pages 196–215, 2005.
- [19] Yongseok Son, Sunggon Kim, Heon Y Yeom, and Hyuck Han. High-performance transaction processing in journaling file systems. In *16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.
- [20] Woong Sul, Kihwang Kim, Minsoo Ryu, Hyungsoo Jung, and Hyuck Han. Fast journaling made simple with NVM. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC)*, pages 1214–1221, 2020.
- [21] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.
- [22] Stephen C Tweedie et al. Journaling the Linux ext2fs filesystem. In *The Fourth Annual Linux Expo*. Durham, North Carolina, 1998.
- [23] Andrew Wilson. The new and improved filebench. In *Proceedings of 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.
- [24] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-enabled IO stack for flash storage. In *16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

Appendix

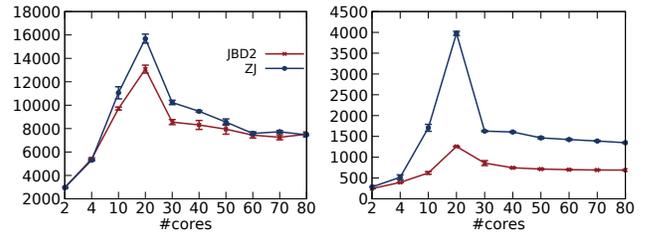
We also evaluated Z-Journal in the data=ordered mode. The performance gain from Z-Journal was less in the ordered mode than in the journal mode because the amount of buffers written to the journals was significantly smaller in the ordered mode. However, the patterns of performance improvement from applying Z-Journal in the ordered mode were similar to that in the journal mode. Figure 10 and Figure 11 show the ordered mode counterparts of Figure 6 and Figure 9, respectively. These are not included in Section 4 for the readability of the graphs and the limited space.



(i) Unlink (low)

(j) Unlink (medium)

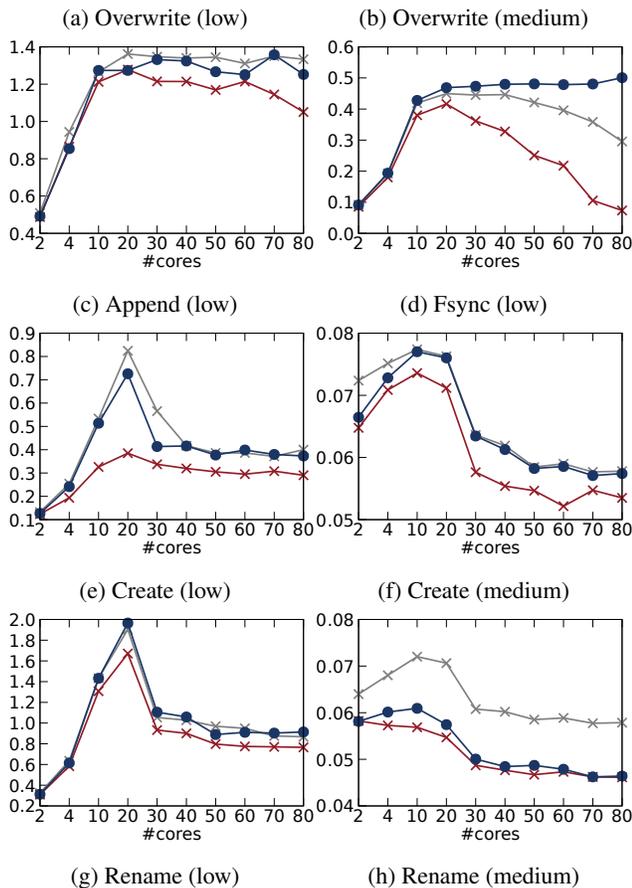
Figure 10: Throughput of FxMark write workloads while varying number of cores under different sharing conditions.



(a) fileserver

(b) varmail

Figure 11: Throughput of Filebench workloads.



(a) Overwrite (low)

(b) Overwrite (medium)

(c) Append (low)

(d) Fsync (low)

(e) Create (low)

(f) Create (medium)

(g) Rename (low)

(h) Rename (medium)

LODIC: Logical Distributed Counting for Scalable File Access

Jeoungahn Park* Taeho Hwang[†] Jongmoo Choi[‡]
Changwoo Min[§] Youjip Won*

*KAIST, Korea [†]Hanyang University, Korea [‡]Dankook University, Korea [§]Virginia Tech, USA

Abstract

We develop a memory-efficient manycore-scalable distributed reference counter for scalable file access, *Logical Distributed Counting* (LODIC). In Logical Distributed Counting, we propose to allocate the local counter on a *per-process* basis. Our process-centric counter design saves the kernel from the excessive memory pressure and the counter query latency issues in the existing *per-core* based distributed counting schemes. The logical distributed counting is designed to dynamically incorporate the three characteristics for reference counting: i) the population of the object, ii) the reference brevity, and iii) the degree of sharing. The key ingredients of the logical distributed counting are *Memory mapping*, *Counter Embedding*, and *Process-space based reverse mapping*. Via mapping a file region to the process address space, LODIC can allocate the local counter at the process address space. With Counter Embedding, the logical distributed counting defines the local counters without the significant changes in the existing kernel code and without introducing significant memory overhead for the local counters. Exploiting the virtual memory segment allocation algorithm of the existing Linux kernel, the process-space based reverse mapping locates the local counter of the physical page without the substantial overhead. Logical Distributed Counting increases the throughput by $65\times$ against stock Linux in reading the shared file block. LODIC exhibits as good performance as the ideal scalable reference counter when deployed in the RocksDB (key value storage) and NGINX (web server) applications.

1 Introduction

Reference counting is a vital part of the modern Operating System (OS). Various kernel objects, e.g. page frame, inode, and file descriptor table maintain the reference counter to prohibit them from being reclaimed prematurely while allowing them to be accessed concurrently by a number of processes without the exclusive lock. The contention on updating the reference counter makes the associated cacheline

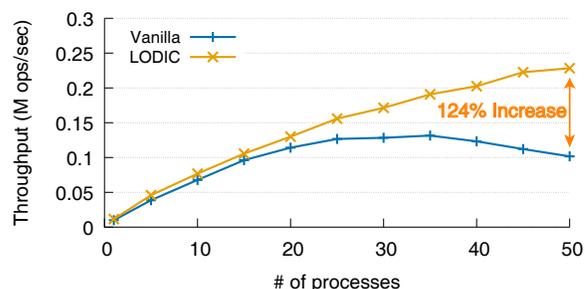


Figure 1: Web server performance accessing the shared web page: wrk benchmark [69] of NGINX web server engine, 120 CPU-cores (Intel Xeon E7-8870 v2 processors, 8 sockets and 15 cores per socket) and 780GB DDR3 DRAM

to be fetched across the CPU cores, and renders the scalability failure due to the cacheline bounce. The importance of the scalable reference counting gets emphasized further as the computer system is loaded with a larger number of CPU cores [17, 23, 31, 40]. Recent demands for the larger main memory in Deep Learning [22], Graph Analysis [48], Virtual Machine consolidation [35, 68], and Big Data Analytics [41] applications make the Operating System to host a larger number of kernel objects and compound the importance of the scalable and the memory-efficient reference counting. Machines with hundreds of the cache coherent cores and the multi-terabytes of the main memory are currently available in the market today: SGI’s Ultra Violet 3000 [10], Dell’s PowerEdge R920 [7], and HPE’s Superdome X servers [37]. In the foreseeable future, we expect a system with thousands of cache coherent low-processing power cores and with even petabytes of main memory [38].

Reading a file block is one of the most essential operations in the computer system, e.g., accessing the web page, searching the database, scanning the key-value file, and etc. Immature implementation of the reference counter for the page frame in the modern Operating Systems makes the shared file block access easily vulnerable to cacheline bounce associated with the reference counter update, leading to scalability and performance collapse [51].

We examine the performance of web server where a popular web page is shared by the number of clients. We increase the number of clients that request for the same web page. Web server daemon deploys multiple processes. Each process services the web page requests from the clients. Fig. 1 illustrates the results. In vanilla Linux that adopts the global reference counting for the page cache entry, the web server throughput saturates at 30 cores. With our logical distributed counting, the web server throughput increases as much as by $2.23\times$ when there are fifty processes. This simple experiment shows that from the application's point of view, reference counting is a vital component in making the shared file block access scalable.

A fair number of works have been proposed to mitigate the contention on the global reference counter [5, 14, 17, 23, 26, 31, 40]. They mitigate the contention on the global counter via allocating the per-core local counters and subsequently via distributing the accesses to the global counter to a number of local counters. They represent a global state of the references using a set of local counters. These works trade the memory pressure and the counter query latency for the scalability for the counter update. To mitigate the memory pressure for the local counters, the recent works propose to allocate the counter cache for each core. The per-core counter cache hosts the recently accessed counters [16, 18, 23, 40].

The existing distributed counter designs are grounded upon the view that the cacheline bounce is caused by the contention among the processors. This processor-centric view on the cacheline contention leads the kernel to define the local counters for each processor core. They blindly define the same number of local counters for all objects regardless of the access popularity. The per-core based distributed counting schemes fail to take into account the actual degree of sharing and impose overly pessimistic estimation on the number of local counters that are required to mitigate the counter contention.

In this work, we view that the cacheline contention in updating the reference counter is driven by the contention among the *processes*, not by the contention among the *processors*. Based upon this new view, we propose to allocate the local counters for a given object in *per-process* basis. We allocate the local counters for a given object to each process that accesses it. We call this distributed counting scheme, *Logical Distributed Counting*, LODIC for short. We call it a *logical* counter since the local counters are associated with the logical entity, the process, not with the physical entity, the processor. In LODIC, the counter query latency is governed by the actual number of processes that share a given file block.

LODIC is designed to address the scalability issue in reading the shared file block [40, 51]. There are three characteristics that need to be incorporated in designing the reference counting scheme: the degree of sharing, the object population, and the reference brevity. Each kernel object, e.g., page frame, dentry entry, inode, and file descriptor table, has widely dif-

ferent reference characteristics along these three axis. The reference counter design should be tailored with respect to the characteristics of the object.

The Logical Distributed Counting consists of three key ingredients: (i) file mapping, (ii) counter embedding, and (iii) process-space based reverse mapping. The first ingredient is to map a file region that needs to be shared to the process address space. This plain and simple mechanism provides a foundation for the logical distributed counting. Via attaching the file-backed page frame to the process address space, LODIC enables the kernel to define the reference counter for the page frame in the process address space, i.e., in per-process basis. The second ingredient is Counter Embedding. With Counter Embedding, LODIC represents the logical counter using the unused bits in page table entry (PTE). Counter Embedding eliminates the need to define a new kernel data structure to represent the local counter and makes the logical counter memory-efficient. The third ingredient is process-space based reverse mapping. Locating the page table of a given page frame is the most expensive task in accessing the local counter in logical distributed counting. For reverse mapping, LODIC scans the virtual memory segments of the process, the *process-space*, unlike the existing reverse mapping feature, `rmap()` [24] that scans the virtual memory segments associated with the file, the *file-space*. The virtual segment allocation algorithm of Linux tends to place the file mapped segments at the high-end of the process virtual address space. Exploiting this nature, the process-space based reverse mapping of LODIC can locate the page table for a given page frame in nearly constant amount of time regardless of the degree of sharing and successfully scales with the degree of sharing. The contribution of LODIC can be summarized as follows.

- Logical Distributed Counting allocates the local counters with respect to the *actual degree of sharing*. The number of local counters for a file block corresponds to the number of processes that map the block which can be substantially smaller than the total number of CPU cores in the large scale manycore system.
- Logical Distributed Counting scheme develops a scalable reverse mapping technique, *process-space based reverse mapping*. It effectively exploits the virtual memory allocation algorithm of existing Linux and can locate the logical counter within a constant amount of time in common cases. The process-space based reverse mapping makes the reverse mapping overhead sufficiently small, whose performance impact associated with accessing the local counter becomes hardly visible from outside.
- Logical Distributed Counting is nearly memory free. The logical counter in LODIC exploits the unused bit space in the page table entry to represent the reference counter. LODIC successfully addresses the memory pressure issue in the logical counting.
- With all these benefits, LODIC increases the performance

of shared block read by $64\times$ compared to the stock Linux. When the file block is shared and accessed by 120 cores.

2 Background and Motivation

2.1 Reference Counting in File Block Access

Page Cache and Reference Counting. OS kernel manages the page cache (or buffer cache) to speed up the file access by caching the frequently accessed disk contents to memory. Linux kernel (and many other OSes) manages per-file (inode) page cache. A page cache maintains the mapping information from a file offset to a physical page frame that holds the associated disk content. In servicing a read/write system call, OS kernel first looks up the page cache to avoid costly storage access. OS kernel tends to keep as much file contents as possible in a page cache. However, under the memory pressure, OS kernel evicts infrequently accessed pages to secure more free memory. One invariant is that OS kernel should not reclaim a page which is still being accessed. Reference counting is commonly used to track the page access count in many OS kernels including Linux. For instance, OS kernel (atomically) increases per-page reference counter before accessing a page in the page cache and decreases it after the access completes. A positive page reference counter means that the page is being accessed and that the page should not be reclaimed.

Page Cache Reference Counting in Linux Linux kernel maintains per-inode page cache (`address_space`) using a radix tree (`XArray`). The page cache maps a file offset to `struct page`, which is metadata on a physical page frame. Linux kernel maintains two atomic reference counter per-page. They are defined in the `struct page`; `_mapcount` representing how many times a page is `mmap`-ed and `_refcount` representing how many tasks are accessing a page [50]. When a file is shared among the multiple processes concurrently, the concurrent update on the `_refcount` becomes the performance bottleneck [51]. When a page frame is chosen for the reclamation, the kernel unlinks it not only from the page cache but also from any page table entries that map a given page cache entry. To quickly locate the associated page table entries, the kernel maintains a reverse mapping from a page frame to a page table entry [24, 25, 43, 67]. Linux kernel maintains an interval tree, which is a set of virtual memory segments mapping a given port of a file to virtual address space (stored in `mapping` field under `struct address_space`).

Challenges in Page Cache Reference Counting. There are two challenges in page cache reference counting. First, the number of reference counters can be prohibitively large. For example, 1 million `_refcounts` are needed to cache 40 GB in Linux. Next, the access characteristics (*e.g.*, access frequency) of a page cache reference counter is determined by the applications that accesses the files, which is out of kernel's control. Therefore, any reference counting scheme for

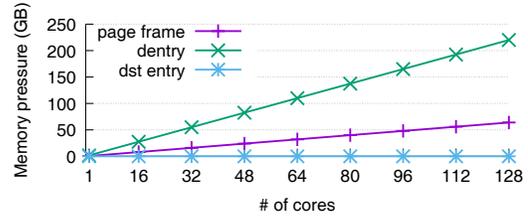


Figure 2: Memory consumption of sloppy counters for page frame (`struct page`, `dentry` and `dst_entry`) when DRAM size is 256GB, the number of `struct page` is 67M, the number of `dentry` is 220M, and the number of `dst_entry` is 32K.

page cache should be memory-efficient to deal with the large memory capacity and should be adaptable to application's file access behavior.

2.2 Scalable Reference Counting

Per-Core Distributed Reference Counting. To avoid the performance collapse caused by cacheline bouncing in atomic counter (used in stock Linux kernel), per-core distributed reference counting schemes – *e.g.*, Scalable Non-Zero Indicator (SNZI) [31], Sloppy Counter [17], and Approximate Counters [26, 27] – have been proposed. This approach manages the per-core local counters in addition to the central object counter. In the common case, each thread accesses its per-core local counter, not incurring cacheline bouncing.

While this approach solves the performance problem of the atomic counter, it has two critical problems that we opted out using it for page cache. First of all, its memory consumption linearly increases as the number of objects and the number of CPU's increase (see Table 1). When a large number of objects (*e.g.*, millions page frames [1,3]) exist in a multi-core machine having tens or hundreds of CPU's, its memory overhead is detrimentally large. For example, Figure 2 shows that sloppy counter requires 60GB of additional memory just to store the reference counters for the page frames (`struct page`) in a 128-core machine with 256GB DRAM (24%). Another drawback is the query performance – getting the true counter value – is often proportional to the number of CPU's. In sloppy counter, the entire local counters should be traversed to get the true value (see Table 1). In managing the page cache, the slow query can slow down the reclamation of page frames and can cause unnecessary swapping in the worst case.

Hash Table Based Reference Counting. To mitigate the excessive memory usage of the per-core schemes, per-core hash table based distributed counting schemes have been proposed. They store local counters at the per-core hash table to bound the memory usage regardless of the number of objects.

However, they do not fundamentally solve the limitation of per-core distributed reference counter. Still their memory consumption is proportional to the number of CPU's as shown in Table 1. Also, when the hash collision happens, they perform its slow path incurring the central contention:

	Atomic Counter	SNZI [31]	Sloppy Counter [17]	RefCache [23]	PayGo [40]	LODIC
Counting Overhead	Contending atomic ops	Non-contending atomic ops	Global lock	Non-atomic ops	Mostly non-atomic ops	Mostly non-contending atomic ops
Space Overhead	$O(N)$	$O(N \cdot C)$	$O(N \cdot C)$	$O(C \cdot H + N)$	$O(C \cdot H + N)$	$O(N)$
Query Overhead	$O(1)$	$O(1)$	$O(C)$	$O(1) + 2 \cdot epoch$	$O(C)$	$O(S)$
Time Overhead	None	None	Every threshold	Every epoch and collision	Every hash collision	None

N : # of objects C : # of CPUs H : size of hash table S : degree of sharing

Table 1: Comparison of reference counting techniques.

for example, RefCache [23] evicts the original entry to the central reference and PayGo [40] evicts to the central overflow counter list. Therefore, unless the hash table size is large enough (*e.g.*, $2 \times$ of active objects), they still suffer from the hash table collision and the slow path execution. Also, they do not improve the query performance because they have to traverse the entire per-core hash tables. In addition to the traversal overhead, RefCache has to wait for two epoch periods until the counter converges, which incurs more delay in getting the true counter value.

3 Design Principles of LODIC

In this section, we present three design principles of LODIC, a scalable reference counting scheme designed for page cache and file access. LODIC should be able to deal with a large number for objects in multi-core machines leveraging the skew of file access and the property of page cache access.

P1. Millions of Counters Are Not Uncommon. Recent advances in multi-core processors and memory technologies make a server equipped with hundreds of cores and hundreds GB to a few TB main memory prevalent. For example, a recent 2-socket Intel Xeon server has up to 112 logical cores and 9 TB memory (3 TB DRAM and 6 TB NVM). We expect more cores with larger memory will be prevalent to achieve higher performance in data-centric applications.

Reference counting should be scalable to the memory capacity and the number of CPU cores. It should scale at least millions of page frames (40 GB memory == 10M reference counters) with a hundred CPU's. To be future-proof, it should scale a few hundred millions (*i.e.*, TB scale) with a few hundreds CPU's. *Ideally, the memory consumption should solely depend on the number of reference counters and the query performance should not depend on the number of cores, which have been increasing.* While the counter caching approach using per-core hash table [23, 40] partly mitigates memory pressure, the current approaches relying on per-core structures [17, 23, 26, 27, 31] are not appropriate in both memory consumption and query performance.

P2. Not All File Pages Are Popular. It is well-known that real-world data access are skewed. Examples include the web page accesses [19], the popularity of the vocabularies in the dictionary [28], and the access distribution in the key-value store [20] to list a few. That implies that the accesses to a small subset of file pages account for dominant fraction of all file accesses while most file pages are not concurrently

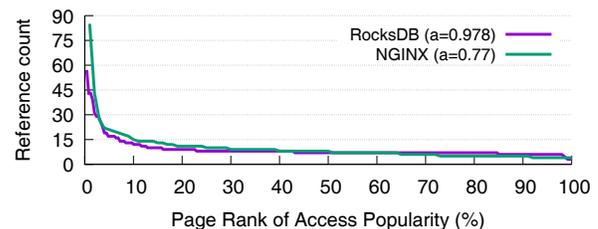


Figure 3: Maximum reference counter distribution of page frames for file access under real-world skewed key-value store and web server accesses on a 120-core server.

accessed at all. The existing distributed reference counting schemes relying on per-core structures [17, 23, 26, 27, 31] fail to exploit such access skew. These works assume that all objects are concurrently accessed from all CPUs. Subsequently, they blindly pre-allocate same number of per-core local counters or same amount of per-core counter cache (hash table) for all objects. The pessimistic assumption on the degree of concurrent access renders overly excessive memory pressure and the query overhead. *Ideally, reference counting scheme should allocate the local counters with respect to the actual degree of sharing, the actual number of processes that share a page.*

We examine the impact of access skew in reference counter design; we run RocksDB key-value store [34] and NGINX web server [60]. For key-value store access, we use the access skew of the English dictionary (Zipf distribution with $\alpha = 0.98$) [28]. For web page access, we use the measurement results for web page access (Zipf distribution with $\alpha = 0.77$) [19]. In this experiment, we record the largest reference counter value in each page frame, which we observed while running the workloads. 100 concurrent clients stress RocksDB having 17,000 key-value pairs with 100-byte values. NGINX runs with 100 concurrent workers. Both were run on a 128-core server. Figure 3 shows the distribution of the maximum reference counter values. X-axis denotes the popularity rank of the pages. Unsurprisingly, the distribution is highly skewed. The sum of the maximum reference counter values for the individual file pages corresponds to the upper bound on the total number of local counters, which are required to represent the concurrent accesses. The sloppy counter [17] allocates 120 local counters for each page. According to this experiment, sloppy counter creates $13.6 \times$ more local counters for RocksDB and $12 \times$ more local counters for NGINX than are actually needed, respectively.

P3. Reference Duration Is Extremely Short in File Access. One important factor in designing the distributed reference

scheme is how to handle the reference split. A process can be scheduled to the different core while the reference is active. Then, the original local counter becomes to reside at the different core from the core where the process is running as a result of the process migration. We call this situation as reference split.

There are two types of approaches to address the reference split problem. The first type of approach is to eliminate the possibility of the reference split. One can temporarily disable the interrupt [47] or the preemption [45] to prevent the process migration. Another type of approach is associated with resolving the reference split when it happens. When the reference split happens, the process can decrease (*i.e.*, unreference) either the local counter at the current core (*e.g.*, RefCache [23]) or the original local counter at the remote core (*e.g.*, PayGo [40]).

Each type of approaches has its own disadvantage; the limited usage, the excessive query latency and the memory overhead. The first is the limited usage. Disabling the interrupt or the preemption cannot be used if the code does not allow to disable the interrupt or the preemption. The second is excessive query latency. RefCache should wait for two epochs to get the real reference counter value. Refcache trades the query latency with the cost of decrementing the counter at the local core. The third is the memory overhead. PayGo requires not only the local counter but also the anchor counter at each core. Also, PayGo needs to maintain the anchor ID at each task struct to handle the reference split.

The above mentioned techniques for the reference split may not deserve its the overhead if the reference split is unlikely to happen. More importantly, all these elaborate schemes deserve its overhead only when the reference split happens frequently. *Therefore, an ideal approach should pay the cost of the reference split only when it really happens.*

To understand how often the reference split can happen in accessing the page cache, we measured the time interval between the reference and the unreference operations for five popular kernel objects including struct page. As Figure 4 shows, the reference duration of page is very short (0.14 μ sec). For page access, the reference split happens very rarely (0.0005%, 5 out of one million page references). To measure the reference duration of page, we repeatedly read 4KB file block for 30 seconds. *Based on our measurement, we carefully conclude that any feature for handling the reference split may hardly deserve its overhead for the reference counting for page cache.*

4 Design of LODIC

4.1 Design Overview

LODIC is designed to scale with a large amount of physical memory and with a large number of CPU's by leveraging the file access skew and the short access duration. We designed

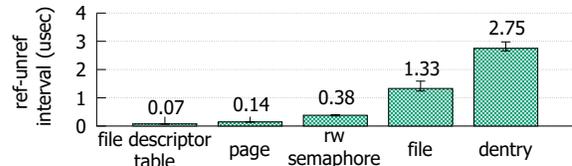


Figure 4: Reference duration for different kernel objects in Linux OS: file descriptor table (struct files_struct), physical page frame (struct page), read-write semaphore (rw_semaphore), file object (struct file) and directory entry (struct dentry).

LODIC in the following three key insights:

First, per-core local counter approaches are by design memory inefficient especially for the file-backed page frames. In file block accesses, the accesses towards a small subset of popular pages account for the dominant fraction of file block accesses as shown in Figure 3. To well exploit the characteristics of the shared file page access, we carefully argue that allocating the local counter in *per-process* basis than in *per-core* basis is appropriate for accessing the page frames. If the local counters are allocated only for the processes which actually share the given file, the memory pressure for the local counters does not increase linearly with CPU-core count but it is only dependent on the actual degree of page sharing. The memory saving of the per-process local counter against the per-core local counter can be particularly substantial in the large scale machine with hundreds of cores since the number of processes that share the given file can be much smaller than the number of CPU cores in the system. Moreover, LODIC reduces the query overhead substantially because it checks the local counters of only those processes that are actually sharing the file. In theory, the number of processes sharing a file can be greater than the number of CPU's. However, we believe such case is rare in practice because, for example, the administrator guides in many servers suggest not to create more (worker) processes beyond the number of CPU's to avoid unnecessary contention on the shared resources [58].

Next, we propose a *selective distributed counting scheme* that exploits the skew in the file accesses. LODIC can selectively apply the distributed counting for the fraction of a file. Not all file blocks in a file are frequently accessed. There may exist a hot region in a file whose reference counting becomes a bottleneck. Thus, we pay the extra overhead of distributed counting when only needed. For instance, the first few levels in B+-tree and log-structured merge (LSM) tree [59] and frequently accessed web pages are the typical examples of the hot file regions. Note that dynamically detecting the hot file blocks are well studied in the prior works [21, 52] and it is out of scope of this paper.

Lastly, we optimize LODIC's *design for the common case that reference split not happening* unlike the prior works [23, 40] preparing handling of the reference split all the time. As shown in §3, the reference split happens extremely rarely.

LODIC exploits two properties of the modern computer system design: (i) there exists a few unused bits left in PTE and

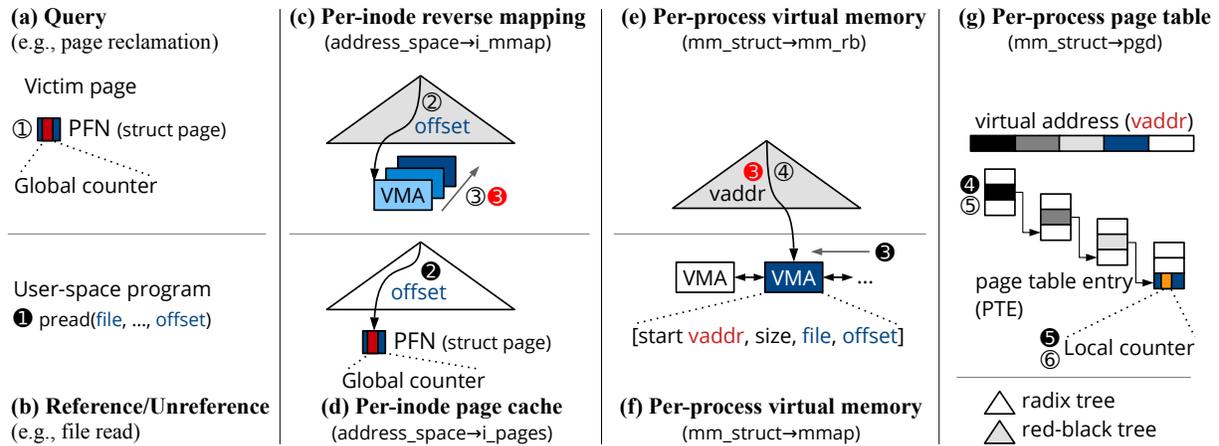


Figure 5: Illustrative examples of LODIC for reference/unreference operations (①) and a query operation (②). ③ is an unoptimized step, which is replaced by LODIC’s optimized step ④ (see the details in §4.4.2).

(ii) Operating System uses the search tree structures to organize the segments in a virtual address space. While the LODIC is currently built on top of Linux, it can be applied to the other Operating Systems. For example, similar to Linux that uses the red-black tree, FreeBSD and Windows use a splay tree and AVL tree, respectively, to manage process address space.

In the next sections, we present three main components of LODIC: (1) per-process selective distributed counting (§4.2), (2) efficient access of a local counter (§4.4), and (3) local counter embedding to page table entry (PTE) (§4.3). We then summarize how these are used for each LODIC operations (§4.5). Figure 5 illustrates the overview of LODIC design for reference/unreference and query operations.

4.2 Selective Distributed Reference Counting

LODIC realizes the per-process distributed counting by mmap-ing a file region to the process virtual address space. By doing so, LODIC allocates the local counters for the virtual pages in the mapped file region. When a program accesses the file block of the mapped file, instead of updating the global reference counter (`_refcount`) in the page frame (`struct page`), LODIC updates the local counters defined for the associated virtual page. The local counter is defined at the process virtual address space. The number of local counters for a given physical page corresponds to the number of virtual pages that map the given physical page frame. This approach deploys the distributed counters for the mapped file regions and for the processes that map a given file.

When a page frame is accessed (①② in Figure 5), LODIC kernel first checks if the given physical page is mmap-ed page or not. If the page is a mmap-ed one, the kernel performs reverse mapping to find a virtual address of the page (③ in Figure 5) then updates the local counter embedded in the associated page table entry (④⑤ in Figure 5). If the page is not mmap-ed (*i.e.*, `_mapcount == 0`) or the contention is low, the kernel falls back to updating the global reference counter in the page

frame. Currently, LODIC determines that the contention on the page is low when the page is shared by less than four processes (*i.e.*, `_mapcount < 4`).

4.3 Local Counter Embedding to PTE

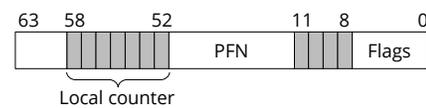


Figure 6: LODIC embeds a local counter to unused bits (colored in gray). In x86-64 architecture, a counter is embedded in bit [58:52].

In allocating the per-process local counter, we propose a *Counter Embedding* technique. With Counter Embedding, LODIC defines the local counter without allocating the separate memory. In Counter Embedding, we represent the per-process local counter using the unused (ignored) bits in the page table entry (PTE). Modern processor architectures leave a few bit in the PTE for software use. For example, x86-64 [12], ARM64 [15], and MIPS64 [53] have 11, 4, and 8 unused bits for software use, respectively. In Figure 6, LODIC in 64 bit x86 architecture is using 7-bit embedded local counter at bits [58:52]. Other CPU technologies also utilize the unused PTE bits (ARM ASID [9], Intel MKTME [39], and AMD SEV [13]). CPU’s with these technologies may leave a fewer bits available for the embedded counter. Regardless of the number of unused bits left in PTE, the design of LODIC remains unchanged. As the local counter embedded at PTE consists of a fewer bits, LODIC will more frequently access the global counter since the embedded counter tends to overflow more frequently. However, in our experimental experience, LODIC rarely accesses the global counter because the local counter overflows infrequently.

The counter embedding has two advantages: First, it does not require any additional memory for the local counter. With counter embedding, LODIC’s space overhead is $O(N)$ as presented in Table 1. In addition, it minimizes the kernel code

changes in implementing the local counter. It does not bring any new data structures nor does it modify any for local counter implementation.

LODIC updates the embedded local counter using an atomic compare-and-swap (CAS) instruction. We decided to use the atomic instruction on purpose. We can simply guarantee the correctness of the local counter update even when the reference split happens without relying on any additional mechanisms such as epoch based counter management in RefCache [23] and anchor counter in PayGo [40]. More importantly, using the atomic instruction does not slow down the common case operation because *non-contending* atomic operations are as cheap as non-atomic operations in modern processor architectures [63]. It is often used in designing the highly optimized lock-free data structures [54]. With 7-bit embedded counter, the embedded local counter overflows when the number of concurrent threads accessing a page is greater than 2^7 , which is extremely rare. To handle the overflow, LODIC falls back to using the global counter to represent the total number of references [61, 65].

4.4 Reverse Mapping from a Physical Page Frame to PTE

The performance and scalability of LODIC critically relies upon the efficiency of the reverse mapping (steps between ② and ④ in Figure 5). If the reverse mapping is slow or becomes a scalability bottleneck in concurrent accesses, it degrades performance significantly. Our evaluation results in §5.2.3 show that the legacy reverse mapping which is used in `rmap()` [24] can degrade the performance by 3 times. We first describe how the existing kernel handles the reverse mapping from a page cache to process virtual address space (§4.4.1) then propose our approach to accelerate the reverse mapping for LODIC (§4.4.2).

4.4.1 Process Space vs. File Space

Process Space: Virtual Address Space. A virtual address space of a process is composed of a set of virtual memory segments. We call this abstraction as *process space*. A virtual memory segment is a virtually contiguous memory region of a process having the same permission and file backend. It is represented by a start virtual address, the size, and backed file and offset (see Figure 5f). A virtual memory segment corresponds to `struct vm_area_struct` in Linux and `struct vm_space` in FreeBSD, respectively. In this paper, we use the segment and the virtual memory area (or VMA), interchangeably. OS kernel uses the process-space structure to check if a given virtual memory access is legal. For example, upon a page fault, OS first checks if the faulting address is legitimate in the process space then handles the fault (e.g., allocating physical page or copy-on-write, etc).

Existing OSes adopt difference data structures for process address space: e.g., red-black tree (Linux) [2], splay tree (FreeBSD) [36], and AVL tree (Windows) [62]. The leaf nodes correspond to the segments. The leaf nodes form a linked list. These data structures have pros and cons. Red-black tree and AVL tree guarantee the worst-case time complexity to $O(\log n)$ for search and update (insert and delete). Splay tree provides amortized time complexity to $O(\log n)$, i.e., the time complexity for k operations is $O(k \log n)$, but the worst-case time complexity of a single operation is worse than the red-black tree or AVL tree. All three OSes rely only on a single lock to protect the process-space structure from race conditions. A number of techniques have been proposed to make the process-space data structures friendly to the concurrent updates [23, 32, 66] to improve the scalability of virtual memory subsystem.

File Space: Page Cache and Reverse Map For each file, the kernel maintains the information associated with a set of the physical pages that cache its file blocks and a set of virtual memory segments that map its file regions if the file (or region of it) is memory mapped. We call this abstraction a *file space*. It corresponds to `struct address_space`¹ in Linux. File space encapsulates two mapping information in it; (i) a mapping from a tuple of (inode, file offset) to a physical page which caches the file block (Figure 5d) and (ii) a mapping from the physical page to the associated page table if the file block is memory mapped. The latter mapping is called *reverse mapping* since it is used to map the physical page to the associated virtual address or the associated page table equivalently. Linux uses a radix tree to organize the set of physical pages in a file space. When the kernel needs to locate the physical page for the given file block (e.g., `read()` and `write()` system calls Figure 5b), the kernel looks up the radix tree of the file space with the given file offset (i.e., `struct address_space` in Figure 5d). In addition, a file can be memory mapped to one or more processes. If the multiple processes map a given file region, the physical page that holds the memory mapped file block can be associated with the multiple process address spaces. To maintain the mappings from the physical page to multiple process address spaces, the kernel maintains the reverse mapping in per-file basis (Figure 5d) in the file space. One example of using the reverse mapping is for page reclamation (Figure 5b). Before the eviction, the kernel should make sure that the page is not being referenced by any threads. The associated page table entries should be invalidated after the page is reclaimed. To quickly invalidate the page table entries of the physical page, the kernel scans the reverse mapping in the file space.

When we map a file region to the process address space, we associate the page frames in a given file region not only to the file space but also to the process space.

¹The name `struct address_space` is, we believe, deeply mis-named.

4.4.2 Accelerating the Reverse Mapping for LODIC

The reverse mapping is the linkage between two spaces: process space and file space. As illustrated in Figure 5c, a reverse mapping (in Linux) maintains a mapping from a file offset to a list of virtual memory segments, which maps the file region. The main usage of the reverse mapping is page reclamation, which happens in every ten of seconds. When a file region is mapped by multiple processes, the kernel needs to examine all segments that map a given file region each of which is associated with different process (③ in Figure 5c) and then to look up the process space to locate the proper virtual memory segment (④ in Figure 5e). Unlike the page reclamation, where the reverse mapping is not in the critical path, LODIC needs to perform the reverse mapping in the critical path at every page cache access to locate the local counter. The existing reverse mapping relying on the file-space data structure does not meet the required level of performance and scalability for LODIC.

LODIC's approach of mapping a file region to the process address enables a new optimization for fast look up of the reverse mapping, which is impossible without mapping the file region. Since the page cache access (i.e., processing read() and write() system calls) is executed in the context of a calling user-space process context, LODIC can directly exploit the process-space data structure (i.e., `current->mm` in Linux) for efficient look up of the reverse mapping (⑤ in Figure 5f).

Leveraging the process-space data structure for the reverse mapping has two important advantages: First, the number of segment to examine is independent of the degree of sharing. Without mapping the file region and leveraging the process-space data structure, LODIC has to traverse the list of the virtual memory segments in the file-space to find out the corresponding virtual address argument. Hence the reverse mapping look up cost linearly increases as the file region is shared by more processes. Next, we can further optimize the reverse mapping look up operation by leveraging the OS's virtual memory layout, which we explain next.

Application	FxMark	RocksDB	NGINX
Number of segments	20	85	62
Position in the segment list	5	4	5

Table 2: Total number of segments in the applications and the position of a memory-mapped file segment in the segment list.

A process virtual address can have tens or hundreds of virtual memory segments. Table 2 illustrates the number of segments in a few popular applications. In reverse mapping, i.e. in locating the page table entry for a given physical page, LODIC exploits the way in which the Linux kernel maps the region of the file onto the process virtual address space (`arch_get_unmapped_area_topdown()`). Exploiting the very nature of kernel's memory mapping, LODIC can quickly locate the target virtual memory segment regardless of the total number of segments in the process virtual ad-

dress space. Linux maps the shared libraries and the memory-mapped files to the virtual address space as follows; when the process starts, the kernel defines the base address in the process virtual address space where the memory mapping starts, `mmap_base`. Starting from the base address, the kernel scans the process virtual address space towards the low end of the virtual address space and looks for the free virtual address region that can accommodate the given size of the file segment (or the shared library). When it finds the free virtual address region that can accommodate the given file segment, it reserves the region of the virtual address space, creates the virtual memory segment object (`struct vma`), and maps the created virtual memory segment to the allocated virtual address region. Due to this mapping mechanism, the virtual memory segment of the memory mapped file is likely to be located just below the stack and the `vDSO` [8]. Table 2 shows the position of the segment for the memory mapped file in the virtual segment list of FxMark, RocksDB and NGINX, respectively. FxMark has twenty virtual memory segments in its virtual address space. From `mmap_base`, the segment for the memory mapped file corresponds to the fifth one in the list of the virtual memory segments.

To leverage such layout property of memory mapped file regions, LODIC scans the segment list of process-space starting from `mmap_base` in reverse direction. By scanning the virtual memory segments in reverse direction (i.e., from high to low virtual addresses), LODIC can quickly find the segment associated with a given page frame (③ in Figure 5). Specifically, LODIC iterates the list of virtual memory segments (`mm_struct->mmmap`) in a reverse direction. It searches a segment that harbors a given page frame. For each segment, the kernel checks if the segment is associated with the same file as the given physical page. If they match, LODIC compares the file offset of the physical page and the file offset of the segment, and check if the page frame belongs to the segment. If we find the associated segment, then we obtain the virtual page number of the given physical page based upon the start virtual address of the segment and the offset of the given page frame within the segment. Once the segment is located and LODIC gets the virtual address of the mapped page, LODIC then walks the page table to locate the embedded local counter (④⑤ in Figure 5). In summary, LODIC can locate the page table entry within nearly constant amount of time independent of the degree of sharing and independent of the number of segments in the virtual address space.

4.5 LODIC Operations

We summarize how LODIC performs each operation with the examples in Figure 5.

Reference Operation. When a page frame in a page cache is accessed (①② in Figure 5), the kernel first performs a reference operation, increasing its reference counter. If the page is memory mapped and it is shared by more than four

processes (*i.e.*, `_mapcount >=4`), LODIC first looks up the corresponding segment by scanning process’s list of virtual memory segments in a reverse order using the file and offset as a key (③). After finding the segment, LODIC calculates the virtual address of the page using the start virtual address of the segment and offset of the mapping. With the virtual address, LODIC walks the page table to spot the page’s local counter (④⑤) and increases the local counter. LODIC relies on atomic CAS to update the page table entry to increase local counter value. If the page is not shared or it is shared by less than four processes, LODIC falls back to increasing page’s global counter (`_refcounter` in `struct page`).

Unreference Operation. After finishing the access of the page, the kernel performs an unreference operation, decreasing the page’s reference counter. The procedure is similar to the reference operation but LODIC handles the situation when the counter changes between the reference and unreference operations. LODIC first tries to decrements the page’s local counter if the page is memory mapped. The local counter can be zero if the number of sharing processes is increased beyond four after the reference operation. To handle such cases, LODIC decrements the global counter when the local counter is zero. Like the reference operation, LODIC relies on atomic CAS to decrement the counter. Since LODIC relies on atomic operation in updating the counter, LODIC does not need any other special mechanism to handle the reference split.

Query Operation. When the kernel reclaims the clean pages in the page cache to secure more free memory, it first performs a query on the page to check if the page is being accessed or not (① in Figure 5). LODIC first looks up the reverse mapping of a file with the file-backed page’s offset and gets the list of virtual memory segments associated with the page (②). For each virtual memory segment (③), LODIC calculates page’s virtual address (④). With the page’s virtual address, LODIC locates the page’s local counter embedded at the PTE (⑤⑥). If any of the page’s global counter and one of the local counters are not zero, LODIC returns the results as non-zero. Otherwise, it return zero so that the kernel can safely reclaim the page. For correctness of a query operation, the local counters and the global counter should remain unchanged while the query is in-flight. To ensure this, we use the same mechanism proposed in [40]; we define a flag for each physical page to denote if there is a query in-flight. Before the query starts, the kernel sets this flag. If this flag is set, the counter update operation blocks. We use `atomic_write` to set and to reset the flag [50].

Latency Analysis of Query Operation. LODIC can return the query operation as soon as it encounters the non-zero local or the global counters since all local counters are guaranteed to be non-negative. This is not possible in some distributed reference counting schemes, such as sloppy counter [17] and RefCache [23]. In these schemes, the local counter can become negative and the positive local counter does not guarantee that

the counter’s true value is positive.

To quantify the performance benefit, we analyze the number of local counter iterations for a query operation. Suppose that there are N counters. We denote the probability that the i -th counter is zero as $P(c_i = 0)$, where c_i is i -th counter value. When X is the number of local counters that the kernel has to examine until encountering a non-zero logical counter, $P(X = k) = p^{k-1}(1 - p)$, where p is $P(c_i = 0)$. Therefore, the expected number of local counter traversal is as follows:

$$E(X_N) = \sum_{i=1}^N i \cdot p^{i-1}(1 - p) = \frac{1 - p^N}{1 - p} \quad (1)$$

$E(X_N)$ grows slowly with N . When $N = 120$ and $p = 0.5$, we obtain $E(X_N) \approx 2$; the kernel examines less than two counters on the average until it encounters a non-zero counter.

5 Evaluation

We implemented LODIC on Linux v4.11.6. We examine the query latency, the memory pressure, and the application performance in five different reference counting schemes: Global Counter (Vanilla Linux), No Counter (NCount), RefCache [23], PayGo [40] and LODIC. We use the microbenchmark FxMark [51] and two data intensive applications – RocksDB key-value store [34] and NGINX web server [60]. Scalable file block read plays an important role in these applications. We choose RocksDB and NGINX in our evaluation because the multi-core servers are widely used in cloud environments, and a key-value store and a web server are the two most popular workloads in the cloud environment. No Counter (NCount) is a null counter that actually does not perform reference counting. This is to simulate the ideal scalable counter. The server has 120 CPU-cores (Intel Xeon E7-8870 v2 processors, 8 sockets and 15 cores per socket) and 780GB DDR3 DRAM. In the rest of this section, we first present how LODIC affects the performance of real-world applications (§5.1) then analyze how our design decisions affect the performance (§5.2).

5.1 Real World Applications

RocksDB. In key-value store, a number of processes can read the same database file, simultaneously. Especially in LSM-tree based key-value store, the run at level 0 can be read by a large number of users simultaneously and the efficiency of the distributed counting scheme can play a vital role in its performance behavior. We use modified `db_bench` [33] so that the multiple processes access the shared database. We perform sequential scan and search on the random key with a key size of 16 byte and a value size of 100 bytes on a working set of 1,000 records. With LODIC, the performance of RocksDB is as good as NCount, RefCache and PayGo (Figure 7). LODIC brings 23% performance improvement against the global counter under 100 processes.

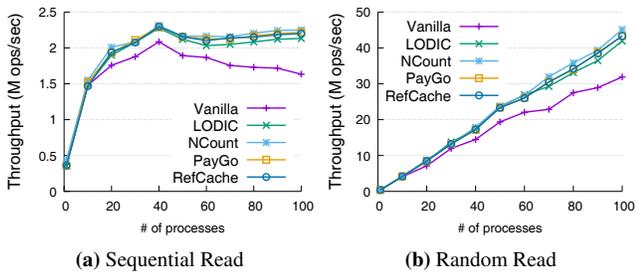


Figure 7: Performance of RocksDB key-value store.

NGINX. We examine the web server performance under the different distributed counting schemes. We vary the number of clients accessing the same web page using wrk benchmark [69]. We vary the number of clients from one to fifty. All clients access `index.html` (612 byte). Figure 8 illustrates the results. The LODIC, NCount, RefCache and PayGo yield the similar performance, showing $2.5\times$ performance improvement against vanilla Linux. Our evaluation results confirm that LODIC performs nearly as good as the ideal contention-less counting scheme (NCount) does.

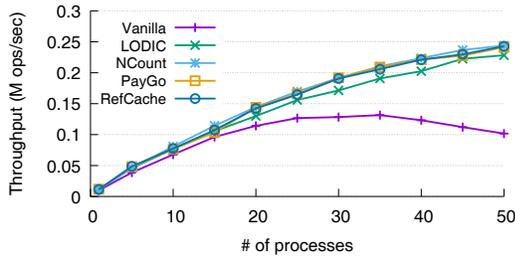


Figure 8: Performance of NGINX web server.

5.2 Analysis on LODIC Design

5.2.1 Memory Pressure

We compute the memory pressure for the RefCache, PayGo and LODIC (Figure 9). In RefCache and PayGo, the memory for distributed counting consists of the memory for the per-core hash tables and the memory for the per-page metadata (Table 1). Refcache and PayGo allocate 64 KB and 256 KB for per-core hash table (4096 entries), respectively. RefCache and PayGo need 16 bytes for each page frame to store the additional information, *e.g.*, lock, pointer, and epoch number. For our server with 780 GB memory (195 M page frames), the additional memory required for all physical page frames corresponds 3.1 GB ($195\text{ M}\times 16\text{ byte}$). Total amount of memory required for the distributed counting in RefCache and in PayGo correspond to 3.12 GB and 3.3 GB, respectively.

In LODIC, the total amount of memory for all local counters corresponds to only the page table pages that are required to map the file (or part of the file). For 4KB, 100MB and 1GB

files, the total amount of memory required for LODIC corresponds to 480KB, 24MB, and 240MB, respectively, when the file is shared by 120 processes. Unlike PayGO and RefCache, LODIC does not have the overhead of handling the hash collision and the cache miss that arise in the counter cache based distributed counting scheme. According to our experiment, as the hash collision increases, the performance of these schemes can drop by 40% from 186 Mops to 70 Mops for 4 KB read.

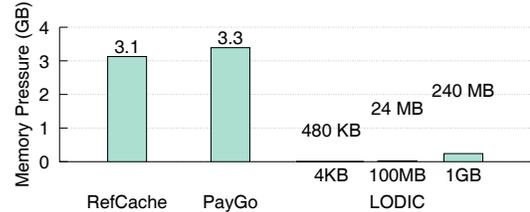


Figure 9: Memory consumption for RefCache, PayGo, and LODIC on a 120-core machine (the degree of sharing in LODIC is 120-core).

5.2.2 Query Overhead

We measure the latency of counter query under two scenarios: (1) the latency to reclaim all page cache entries for a given file (Figure 10) and (2) the latency to check if a given page frame can be reclaimed (Figure 11).

We examine the latency to reclaim all page cache entries of the 1 GB file under vanilla Linux (Vanilla), RefCache, PayGo, and LODIC. We use `fadvice` to trigger the page reclamation. In LODIC, we vary the fraction of the mapped pages (*i.e.*, hot page ratio) in the file; 10%, 20% and 100%. Figure 10 illustrates the results. ‘Vanilla’ renders the best case. Here, the kernel examines only the single global counter for each page frame to see if it can be reclaimed. The kernel repeats this process for all page frames for the given file. In per-core distributed counting schemes, the latency of reclaiming all page frames is problematic. It examines all per-core local counters for all pages in the file. The latency of `fadvice` corresponds to over 500 msec regardless of the number of processes that share the file. For the page reclamation, the LODIC uses the red-black tree (3 in Figure 5c) stored in `i_mmap` field of `struct address_space` like the Vanilla. In LODIC, the number of the local counters for a page frame is proportional to the number of the processes that map the file. For unmapped page, there is no local counter. The query latency of LODIC is longer than the vanilla Linux. However, LODIC successfully reduces the query latency properly incorporating the actual degree of sharing. When 10% of the entire file is hot region (100 MB is mapped), LODIC renders less than 1/4 of the `fadvice` latency of the per-core distributed counter when the file is shared by 45 processes. However, in the worst case (*i.e.*, the entire file is hot), which we believe unrealistic, the query latency of LODIC exceeds the query latency of the per-core distributed counting scheme when the number of processes is greater than 40. This is due to the overhead of reverse mapping in LODIC.

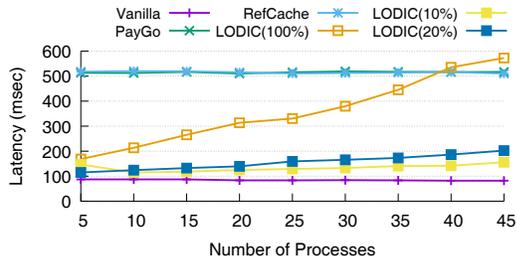


Figure 10: Query latency: reclaiming all page frames (fadvice).

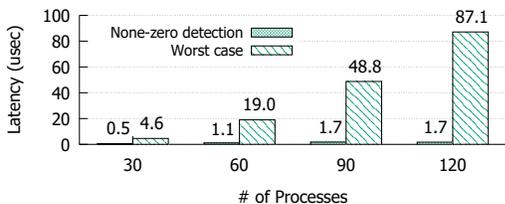


Figure 11: Query latency: reclaiming a page frame.

For counter query, we also measure the latency to determine if a given page is reclaimable. In LODIC, the kernel returns as soon as it encounters the non-zero local counter. As Figure 11 shows, LODIC examines only one or two local counters in most cases until it encounters non-zero local counter. The result well matches Equation 1. In the worst case, LODIC needs to scan all logical counters till it finds the non-zero local counter. In this case, the latency linearly increases with the degree of sharing.

5.2.3 Reverse Mapping Overhead

Performance and Scalability. We first examine the efficiency of the file-space based reverse mapping (3) in Figure 5) and the process-space based reverse mapping (3) in Figure 5). Figure 12 shows the time to perform the reverse mapping under two different reverse mapping schemes. The reverse mapping latency reduces to 1/20 when we use the process-space based reverse mapping instead of the file-space based reverse mapping.

Next, we examine the performance impact of the reverse mapping. We compare the latency to read 4 KB block from the different regions of a file: (1) from plain file in Vanilla Linux, (2) from the unmapped region of a file in LODIC-enabled Linux, and (3) from the memory-mapped region of the file

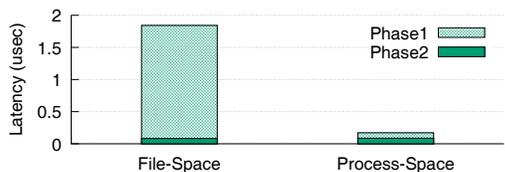


Figure 12: Latency break-down: file-space based reverse mapping vs. process-space based reverse mapping

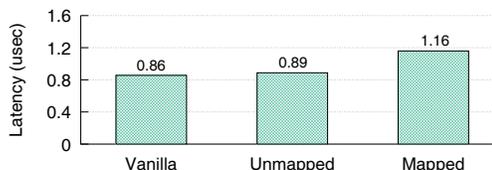


Figure 13: Read latency with various reverse mapping schemes.

in LODIC-enabled Linux. Figure 13 illustrates the result. In LODIC-enabled Linux, reading the memory-mapped file block renders 20% longer latency than reading the unmapped file block. In LODIC, the filesystem checks the map count of the file block before it performs reverse mapping. It performs the reverse mapping only when the map count is non-zero.

Lastly, we examine the performance of reading a shared file block under different reference counting schemes: (1) process-space based reverse mapping (LODIC), (2) file-space based reverse mapping (LODIC) and (3) vanilla Linux. We use DRBH workload of FxMark [51]. Figure 14 presents the results. Using the process-space based reverse mapping, LODIC achieves 65x performance against vanilla Linux under 120 CPU cores. The benchmark performance successfully scales linearly with respect to the increase in the number of processes (i.e., the number of active cores). On the other hand, the file-space based reverse mapping fails to scale due to its inefficient access path shown at 3) in Figure 5.

Different from the rests, XFS barely yields any performance gain under LODIC. We find that reference counter in per-inode rw-semaphore in XFS is the root cause of the scalability bottleneck here (Figure 14d). The performance bottleneck caused by this problematic inode reference counter has been out-shadowed by the severe scalability overhead of the reference counter for physical page. As LODIC effectively resolves the scalability issue in the reference counter of the physical page, the inefficiency of inode reference counter in XFS comes to the surface and prohibits XFS from scaling well. LODIC, however, is not entirely ineffective; LODIC brings as much as 2x performance in XFS (Figure 14d).

Interference with mmap. The mmap and munmap operation establish an exclusive lock on the process-space and the file-space structure to insert and delete the virtual memory segment. We examine how the mmap and munmap operations interfere with the performance of LODIC. There are 60 processes that read the shared file block. We create a background process that calls mmap, sleeps for one second and does munmap. We measure the performance of the shared file read with and without the background processes, respectively, and compute the performance ratio between the two.

Figure 15 shows the performance degradation varying the number of background processes. The performance of process-space based reverse mapping is barely affected by the background mmap/munmap. On the other hand, the performance of the file-space based reverse-mapping collapses to 1/100 when there are five or more background processes.

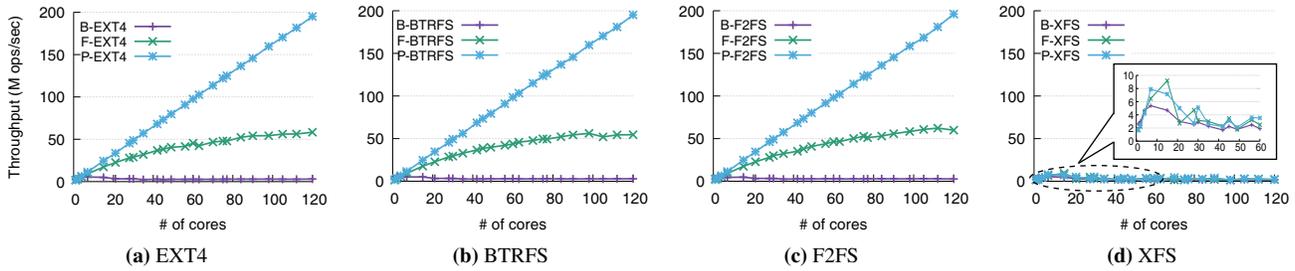


Figure 14: FxMark (DRBH): Baseline ('B'), File-based reverse mapping ('F'), Process-based reverse mapping ('P').

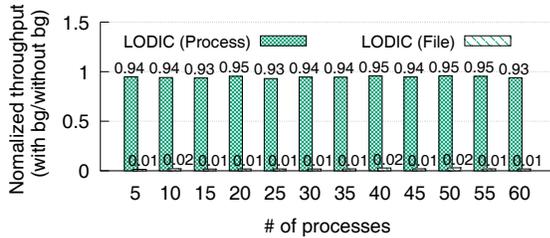


Figure 15: mmap() interference on the reverse mapping.

5.2.4 Counter Contention on the Local Counter

We examine the theoretical worst-case performance of LODIC; when the threads compete for updating the local counter. We create multiple threads in a process and have them access the same file block. We vary the number of threads per process to vary the contention on the local counters. We maintain the total number of threads in the system to be 120. Table 3 illustrates the performance. As the number of threads per process increases, the performance decreases. This is because PTE contention occurs when many threads of a process access the same file page at the same time. However, compared to the case using the global counter, LODIC is meaningful in that the contention is localized within a process because a PTE is shared by threads only within a process. We believe that in practice this situation can be avoided via properly limiting the number of threads per process. For example, it is common for server programs to provide tuning knobs to change the number of threads per process [56–58].

# of Threads	1	2	4	8	16
Throughput (M ops/sec)	164.16	58.87	32.54	19.61	8.79

Table 3: Effect of the contention on the local counters

6 Related Work

There were a number of sophisticated designs for the scalable reader-write lock. For the scalability, a few works exploit the memory barrier and the atomic instruction in the reader [44, 64]. Big-reader lock (brlock) uses an array of reader flags to avoid using the memory barrier [4]. Passive reader-write lock (prwlock) uses version-based con-

ensus protocol instead of the expensive mutex protected flags to avoid using the memory barrier [46]. A new reader-writer lock called percpu-rwlock in Linux [6] is known to work when the writers are rare. Bravo [30] uses the global hash table to address the memory pressure issue of the per-core reference counters [6, 17, 26, 27, 31] for rw-semaphore. SHFLLOCK [42] leverages a waiting thread to achieve high scalability in NUMA machines without introducing the additional memory consumption for NUMA-local locks. Hydralist separates the search layer and the data node for the manycore scalability of the index structure [49].

A few works adopt the actual degree of sharing to ease the contention [11, 29, 55]. Nerula et al. [55] detect the access conflict at run-time and distribute the conflict records to per-core basis. It cannot be applied to the generic workload. Dashti et al. [29] use the hardware profiling to detect the degree of sharing. It fails to scale when the sharing degree is high.

7 Conclusion

The per-core based distributed counting scheme has reached its limit due to the increase in the number of cores and the memory size in the recent computing system. In this work, we view that the counter contention is driven by the contention among the processes. This process-centric view enables us to devise a new counting scheme that can naturally incorporate the degree of sharing, the population and the reference brevity characteristics of the physical page frame. Via defining the local counter in per-process basis, LODIC is successful in striking the balance among the three factors of the reference counter: memory pressure, the counter query latency, and counter update performance. We show that software overhead of the proposed logical distributed counting is insignificant and hardly visible from outside.

Acknowledgements We would like to thank our shepherd James Bottomley and the anonymous reviewers for their valuable feedback. We also would like to thank Yeonjin Noh and Hyeongjun Kim for their help on the earlier draft of this work. This work was supported by IITP, Korea (grant No. 2018-0-00549 and No. 2014-3-00035) and NRF, Korea (grant No. NRF-2020R1A2C3008525).

References

- [1] 10M Concurrent Websockets. <http://goroutines.com/10m>.
- [2] Linux rbtree. <https://www.kernel.org/doc/Documentation/rbtree.txt>.
- [3] Number of dentry objects. <https://serverfault.com/questions/561350/unusually-high-dentry-cache-usage>.
- [4] Big reader locks. <https://lwn.net/Articles/378911/>, 2010.
- [5] The search for fast, scalable counters. <http://lwn.net/Articles/170003/>, 2010.
- [6] percpu_rwlock: Implement the core design of per-cpu reader-writer locks. <https://lore.kernel.org/patchwork/patch/360375/>, 2013.
- [7] Dell poweredge specification. http://media.zones.com/images/pdf/Dell_PowerEdge_R920.pdf, 2014.
- [8] vdso. <https://lwn.net/Articles/615809/>, 2014.
- [9] Address space identifier. <https://lwn.net/Articles/699820/>, 2016.
- [10] Sgi ultraviolet 3000. https://www.sgi.com/products/servers/uv/uv_3000_30.html, 2016.
- [11] Umut A Acar, Naama Ben-David, and Mike Rainey. Contention in structured concurrency: Provably efficient dynamic non-zero indicators for nested parallelism. *ACM SIGPLAN Notices*, 52(8):75–88, 2017.
- [12] AMD. Amd64 architecture programmer’s manual volume 2: System programming.
- [13] AMD. Secure encrypted virtualization. <https://developer.amd.com/sev/>.
- [14] Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, Amos Waterland, Robert W Wisniewski, Jimi Xenidis, Michael Stumm, et al. Experience distributing objects in an smmp os. *Transactions on Computer Systems (TOCS)*, 25(3):6, 2007.
- [15] ARM. Arm® architecture reference manual armv8, for armv8-a architecture profile.
- [16] 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 *Proc. of ACM SOSP*, 2017.
- [17] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Morris, Nickolai Zeldovich, et al. An analysis of linux scalability to many cores. In *Proc. of USENIX OSDI*, 2010.
- [18] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Opllog: a library for scaling update-heavy data structures. *Technical Report MIT-CSAIL-TR-2014-019*, 2014.
- [19] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proc. of IEEE INFOCOM*, 1999.
- [20] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *Proc. of USENIX FAST*, 2020.
- [21] Mei-Ling Chiang, Paul CH Lee, and Ruei-Chuan Chang. Using data clustering to improve cleaning performance for flash memory. *Software: Practice and Experience*, 29(3):267–290, 1999.
- [22] Wonje Choi, Karthi Duraisamy, Ryan Gary Kim, Janardhan Rao Doppa, Partha Pratim Pande, Radu Marculescu, and Diana Marculescu. Hybrid network-on-chip architectures for accelerating deep learning kernels on heterogeneous manycore platforms. In *Proc. of IEEE CASES*, 2016.
- [23] Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. Radixvm: Scalable address spaces for multithreaded applications. In *Proc. of ACM EUROSYS*, 2013.
- [24] Corbet. The object-based reverse-mapping vm. <https://lwn.net/Articles/23732/>, Feb 2003.
- [25] Corbet. Kswapd and high-order allocations. <https://lwn.net/Articles/101230/>, Sep 2004.
- [26] J Corbet. Per-cpu reference counts. <https://lwn.net/Articles/557478/>, 2013.
- [27] J Corbet. The search for fast, scalable counters. <https://lwn.net/Articles/170003/>, 2016.
- [28] Alvaro Corral, Gemma Boleda, and Ramon Ferrer-i Cancho. Zipf’s law for word frequencies: Word forms versus lemmas in long texts. *PLOS ONE*, 10(7), 2015.
- [29] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: a holistic approach to memory placement on numa systems. *ACM SIGARCH Computer Architecture News*, 41(1):381–394, 2013.

- [30] Dave Dice and Alex Kogan. Bravo: Biased locking for reader-writer locks. In *Proc. of USENIX ATC*, 2019.
- [31] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. Snzi: Scalable nonzero indicators. In *Proc. of ACM PODC*, 2007.
- [32] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the BSDCan Conference*, 2006.
- [33] Facebook. db_bench. <https://github.com/facebook/rocksdb/wiki/Benchmarking-tools>.
- [34] Facebook. RocksDB. <http://rocksdb.org>.
- [35] Md Hasanul Ferdaus, Manzur Murshed, Rodrigo N Calheiros, and Rajkumar Buyya. Virtual machine consolidation in cloud data centers using aco metaheuristic. In *Proc. of EuroPar*, 2014.
- [36] FreeBSD. <https://www.freebsd.org>.
- [37] HPE. Hpe integrity superdome x. <https://www.hpe.com/us/en/servers/superdome.html>, 2016.
- [38] HPE. Hpe the machine. <http://www.labs.hpe.com/research/themachine/>, 2016.
- [39] Intel. Multi-key total memory encryption. <https://software.intel.com/content/dam/develop/external/us/en/documents-tps/multi-key-total-memory-encryption-spec.pdf>.
- [40] Seokyoung Jung, Jongbin Kim, Minsoo Ryu, Sooyong Kang, and Hyungsoo Jung. Pay migration tax to homeland: Anchor-based scalable reference counting for multicores. In *Proc. of USENIX FAST*, 2019.
- [41] Karim Kanoun, Martino Ruggiero, David Atienza, and Mihaela Van Der Schaar. Low power and scalable many-core architecture for big-data stream computing. In *Proc. of IEEE ISVLSI*, 2014.
- [42] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. Scalable and practical locking with shuffling. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 586–599. ACM, 2019.
- [43] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proc. of Linux Symposium*, 2007.
- [44] Yossi Lev, Victor Luchangco, and Marek Olszewski. Scalable reader-writer locks. In *Proc. of ACM SPAA*, 2009.
- [45] linux. Proper locking under a preemptible kernel: Keeping kernel code preempt-safe. <https://www.kernel.org/doc/Documentation/preempt-locking.txt>.
- [46] Ran Liu, Heng Zhang, and Haibo Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *Proc. of USENIX ATC*, 2014.
- [47] lwn. Linux generic irq handling. <https://static.lwn.net/kerneldoc/core-api/genericirq.html>.
- [48] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proc. of ACM EUROSYS*, 2017.
- [49] Ajit Mathew and Changwoo Min. Hydralist: A scalable in-memory index using asynchronous updates and partial replication. *Proc. VLDB Endow.*, 13(9):1332–1345, 2020.
- [50] Paul E McKenney. Overview of linux-kernel reference counting, 2007.
- [51] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *Proc. of USENIX ATC*, 2016.
- [52] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: random write considered harmful in solid state drives. In William J. Bolosky and Jason Flinn, editors, *Proc. of USENIX FAST*, page 12. USENIX Association, 2012.
- [53] MIPS. Mips® architecture for programmers volume iii: Mips64® / micromips64™ privileged resource architecture.
- [54] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '13, Shenzhen, China, February 23-27, 2013*, pages 103–112. ACM, 2013.
- [55] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase reconciliation for contended in-memory transactions. In *Proc. of USENIX OSDI*, 2014.
- [56] NGINX. Apache performance tuning: Mpm directives. <https://www.liquidweb.com/kb/apache-performance-tuning-mpm-directives/>.
- [57] NGINX. Thread pools in nginx boost performance 9x! <https://www.nginx.com/blog/thread-pools-boost-performance-9x/>.

- [58] NGINX. Tuning nginx for performance. <https://www.nginx.com/blog/tuning-nginx/>.
- [59] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514, 2017.
- [60] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.
- [61] Kenneth Russell and David Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. *ACM SIGPLAN Notices*, 41(10):263–272, 2006.
- [62] M.E. Russinovich, D.A. Solomon, and A. Ionescu. *Windows Internals*. Pearson Education, 2012.
- [63] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architectures and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*, pages 445–456. IEEE Computer Society, 2015.
- [64] Michael L Scott and John M Mellor-Crummey. Synchronization without contention. In *Proc. of ASPLOS*, 1991.
- [65] Rifat Shahriyar, Stephen M Blackburn, and Daniel Frampton. Down for the count? getting reference counting back in the ring. In *Proceedings of the 2012 international symposium on Memory Management*, pages 73–84, 2012.
- [66] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: The continuously concurrent compacting collector. *ACM SIGPLAN Notices*, 46(11):79–88, 2011.
- [67] Rik Van Riel. Page replacement in linux 2.4 memory management. In *Proc. of USENIX ATC*, 2001.
- [68] Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari. Server workload analysis for power minimization using consolidation. In *Proc. of USENIX ATC*, 2009.
- [69] Modern http benchmarking tool, 2013. <https://github.com/wg/wrk>.

UNISTORE: A fault-tolerant marriage of causal and strong consistency

Manuel Bravo

Alexey Gotsman

Borja de Régil

Hengfeng Wei *

IMDEA Software Institute

Nanjing University

Abstract

Modern online services rely on data stores that replicate their data across geographically distributed data centers. Providing strong consistency in such data stores results in high latencies and makes the system vulnerable to network partitions. The alternative of relaxing consistency violates crucial correctness properties. A compromise is to allow multiple consistency levels to coexist in the data store. In this paper we present UNISTORE, the first fault-tolerant and scalable data store that combines causal and strong consistency. The key challenge we address in UNISTORE is to maintain liveness despite data center failures: this could be compromised if a strong transaction takes a dependency on a causal transaction that is later lost because of a failure. UNISTORE ensures that such situations do not arise while paying the cost of durability for causal transactions only when necessary. We evaluate UNISTORE on Amazon EC2 using both microbenchmarks and a sample application. Our results show that UNISTORE effectively and scalably combines causal and strong consistency.

1 Introduction

Many of today's Internet services rely on geo-distributed data stores, which replicate data in different geographical locations. This improves user experience by allowing accesses to the closest site and ensures disaster-tolerance. However, geo-distribution also makes it more challenging to keep the data consistent. The classical approach is to make replication transparent to clients by providing strong consistency models, such as linearizability [33] or serializability [70]. The downside is that this approach requires synchronization between data centers in the critical path. This significantly increases latency [1] and makes the system unavailable during network partitionings [28]. Thus, even though several commercial geo-distributed systems follow this approach [18, 20, 26, 63, 71], the associated cost has prevented it from being adopted more widely.

An alternative approach is to relax synchronization: the data store executes an operation at a single data center, without any communication with others, and propagates updates to other data centers in the background [21, 66]. This minimizes the latency and makes the system *highly available*, i.e., operational even during network partitionings. But on the downside, the systems following this approach provide weaker consistency models: e.g., eventual consistency [66, 69] or *causal consistency* [2]. The latter is particularly appealing: it guarantees that clients see updates in an order that respects the potential causality between them. For example, assume that in a banking application Alice deposits \$100 into Bob's account (u_1) and then posts a notification about it into Bob's inbox (u_2). Under causal consistency, if Bob sees the notification (u_3), and then checks his account balance (u_4), he will see the deposit. This is not guaranteed under eventual consistency, which does not respect causality relationships, such as those between u_1 and u_2 . In some settings, causal consistency has been shown to be the strongest model that allows availability during network partitionings [7, 45]. It has been a subject of active research in recent years, with scalable implementations [3, 43, 48] and some industrial deployments [55, 67].

However, even causal consistency is often too weak to preserve critical application invariants. For example, consider a banking application that disallows overdrafts and thus maintains an invariant that an account balance is always non-negative. Assume that the balance of an account stored at two replicas is 100, and clients concurrently issue two `withdraw(100)` operations (u_5 and u_6) at different replicas. Since causal consistency executes operations without synchronization, both withdrawals will succeed, and once the replicas exchange the updates, the balance will go negative. To ensure integrity invariants in examples such as this, the programmer has to introduce synchronization between replicas, and, since synchronization is expensive, it pays off to do this sparingly. To this end, several research [9, 41, 42, 65] and commercial [5, 6, 29, 49, 58] data stores allow the programmer to choose whether to execute a particular operation under weak or strong consistency. For example, to preserve the integrity

*Also with the State Key Laboratory for Novel Software Technology, Software Institute.

invariant in our banking application, only withdrawals need to use strong consistency, and hence, synchronize; deposits may use weaker consistency and proceed without synchronization.

Given the benefits of causal consistency, it is particularly appealing to marry it with strong consistency in a geo-distributed data store. But like real-life marriages, to be successful this one needs to hold together both in good times and in bad – when data centers fail due to catastrophic events or power outages. Unfortunately, none of the existing data stores meant for geo-replication combine causal and strong consistency while providing such fault tolerance [9, 41, 42]. In this paper we present UNISTORE – the first fault-tolerant and scalable data store that combines causal and strong consistency. More precisely, UNISTORE implements a transactional variant of *Partial Order-Restrictions consistency (PoR consistency)* [31, 42]. This guarantees transactional causal consistency by default [3] and allows the programmer to additionally specify which pairs of transactions *conflict*, i.e., have to synchronize. For instance, to preserve the integrity invariant in our previous example, the programmer should declare that withdrawals from the same account conflict. Then one of the withdrawals u_5 and u_6 will observe the other and will fail.

The key challenge we have to address in UNISTORE is to maintain liveness despite data center failures. Just adding a Paxos-based commit protocol for strong transactions [19, 20, 35] to an existing causally consistent protocol does not yield a fault-tolerant data store. In such a data store, a committed strong transaction t_2 may never become visible to clients if a causal transaction t_1 on which it depends is lost due to a failure of its origin data center. This compromises the liveness of the system, because no transaction t_3 conflicting with t_2 can commit from now on: according to the PoR model, one of the transactions t_2 and t_3 has to observe the other, but t_2 will never be visible and t_2 did not observe t_3 .

UNISTORE addresses this problem by ensuring that, before a strong transaction commits, all its causal dependencies are *uniform*, i.e., will eventually become visible at all correct data centers. This adapts the classical notion of uniformity in distributed computing to causal consistency [16]. UNISTORE does so without defeating the benefits of causal consistency. Causal transactions remain highly available at the cost of increasing the latency of strong transactions: a strong transaction may have to wait for some of its dependencies to become uniform before committing. To minimize this cost, UNISTORE executes causal transactions on a snapshot that is slightly in the past, such that a strong transaction will mostly depend on causal transactions that are already uniform before committing. Furthermore, UNISTORE reuses the mechanism for tracking uniformity to let clients make causal transaction durable on demand and to enable consistent client migration.

In addition to being fault tolerant, UNISTORE scales horizontally, i.e., with the number of machines in each data center; this also goes beyond previous proposals [9, 41, 42]. To this end, UNISTORE builds on Cure [3] – a scalable implementa-

tion of transactional causal consistency. Our protocol extends Cure with a novel mechanism that distributes the task of tracking the set of uniform transactions among the machines of a data center. We also add the ability for data centers to forward transactions they receive from others, so that a transaction can propagate through the system even if its origin data center fails. Finally, we carefully integrate an existing fault-tolerant atomic commit for strong transactions [19] into the protocol for causal consistency.

We have rigorously proved the correctness of the UNISTORE protocol (§7 and [12, §D]). We have also evaluated it on Amazon EC2 using both microbenchmarks and a more realistic RUBiS benchmark. Our evaluation demonstrates that UNISTORE scalably combines causal and strong transactions, with the latter not affecting the performance of the former. Under the RUBiS mix workload, causal transactions exhibit a low latency (1.2ms on average), and the overall average latency is $3.7\times$ lower than that of a strongly consistent system.

2 System Model

We consider a geo-distributed system consisting of a set of data centers $\mathcal{D} = \{1, \dots, D\}$ that manage a large set of data items. A data item is uniquely identified by its *key*. For scalability, the key space is split into a set of logical partitions $\mathcal{P} = \{1, \dots, N\}$. Each data center stores replicas of all partitions, scattered among its servers. We let p_d^m be the replica of partition m at data center d , and we refer to replicas of the same partition as *sibling* replicas. As is standard, we assume that $D = 2f + 1$ and at most f data centers may fail. We call a data center that does not fail *correct*. If a data center fails, all partition replicas it stores become unavailable. For simplicity, we do not consider the failures of individual replicas within a data center: these can be masked using standard state-machine replication protocols executing within a data center [38, 56].

Replicas have physical clocks, which are loosely synchronized by a protocol such as NTP. The correctness of UNISTORE does not depend on the precision of clock synchronization, but large drifts may negatively impact its performance. Any two replicas are connected by a reliable FIFO channel, so that messages between correct data centers are guaranteed to be delivered. As is standard, to implement strong consistency we require the network to be *eventually synchronous*, so that message delays between sibling replicas in correct data centers are eventually bounded by some constant [24].

3 Consistency Model

A client interacts with UNISTORE by executing a stream of *transactions* at the data center it is connected to. A transaction consists of a sequence of operations, each on a single data item, and can be *interactive*: the data items it accesses are not known a priori. A transaction that modifies at least one data item is an *update* transaction; otherwise it is *read-only*.

A *consistency model* defines a contract between the data store and its clients that specifies which values the data store is allowed to return in response to client operations. UNISTORE implements a transactional variant of *Partial Order-Restrictions consistency (PoR consistency)* [31, 42], which we now define informally; we give a formal definition in [12, §B]. The PoR model enables the programmer to classify transactions as either *causal* or *strong*. Causal transactions satisfy transactional *causal consistency*, which guarantees that clients see transactions in an order that respects the potential causality between them [2, 3]. However, clients can observe causally independent transactions in arbitrary order. Strong transactions give the programmer more control over their visibility. To this end, the programmer provides a symmetric *conflict relation* \bowtie on operations that is lifted to strong transactions as follows: two transactions conflict if they perform conflicting operations on the same data item. Then the PoR model guarantees that, out of two conflicting strong transactions, one has to observe the other.

More precisely, a transaction t_1 precedes a transaction t_2 in the *session order* if they are executed by the same client and t_1 is executed before t_2 . A set of transactions T committed by the data store satisfies PoR consistency if there exists a *causal order relation* \prec on T such that the following properties hold:

Causality Preservation. The relation \prec is transitive, irreflexive, and includes the session order.

Return Value Consistency. Consider an operation u on a data item k in a transaction $t \in T$. The return value of u can be computed from the state of k obtained as follows: first execute all operations on k by transactions preceding t in \prec in an order consistent with \prec ; then execute all operations on k that precede u in t .

Conflict Ordering. For any distinct strong transactions $t_1, t_2 \in T$, if $t_1 \bowtie t_2$, then either $t_1 \prec t_2$ or $t_2 \prec t_1$.

Eventual Visibility. A transaction $t \in T$ that is either strong or originates at a correct data center eventually becomes visible at all correct data centers: from some point on, t precedes in \prec all transactions issued at correct data centers.

If all transactions are causal, then the above definition specializes to transactional causal consistency [3, 17]. If all transactions are strong and all pairs of operations conflict, then we obtain (non-strict) serializability.

When $t_1 \prec t_2$, we say that t_1 is a *causal dependency* of t_2 . Return Value Consistency ensures that all operations in a transaction t execute on a snapshot consisting of its causal dependencies (as well as prior operations by t). Transactions are atomic, so that either all of their operations are included into the snapshot or none at all. The transitivity of \prec , mandated by Causality Preservation, ensures that the snapshot a transaction executes on is *causally consistent*: if a transaction t_1 is included into the snapshot, then so is any other transaction t_2 on which t_1 depends (i.e., $t_2 \prec t_1$). The inclusion of the session order into \prec , also mandated by Causality Preservation,

ensures session guarantees such as *read your writes* [64]. The consistency model disallows the causality violation anomaly from §1. Indeed, since \prec includes the session order, we have $u_1 \prec u_2$ and $u_3 \prec u_4$. Moreover, Bob sees Alice’s message, and by Return Value Consistency this can only happen if $u_2 \prec u_3$. Then since \prec is transitive, $u_1 \prec u_4$, and by Return Value Consistency, Bob has to see Alice’s deposit.

Causal consistency nevertheless allows the overdraft anomaly from §1: the withdrawals u_5 and u_6 may not be related by \prec , and thus may both execute on the balance 100 and succeed. The Conflict Ordering property can be used to disallow this anomaly by declaring that `withdraw` operations on the same account conflict and labeling transactions containing these as strong. Then one of the withdrawal transactions will be guaranteed to causally precede the other. The latter will be executed on the account balance 0 and will fail.

Finally, Eventual Visibility ensures that strong transactions and those causal ones that originate at correct data centers are durable, i.e., will eventually propagate through the system.

To facilitate the use of causal transactions, UNISTORE includes *replicated data types* (aka CRDTs), which implement policies for merging concurrent updates to the same data item [57]. Each data item in the store is associated with a type (e.g., counter, set), which is backed by a CRDT implementation managing updates to it. For example, the programmer can use a counter CRDT to represent an account balance. Then if two clients concurrently deposit 100 and 200 into an empty account using causal transactions, eventually the balance at all replicas will be 300. Using ordinary writes here would yield 100 or 200, depending on the order in which the writes are applied. More generally, CRDTs ensure that two replicas receiving the same set of updates are in the same state, regardless of the receipt order. Together with Eventual Visibility, this implies the expected guarantee of eventual consistency [66]. Due to space constraints, we omit details about the use of CRDTs from our protocol descriptions.

4 Key Design Decisions in UNISTORE

Baseline causal consistency. A causal transaction in UNISTORE first executes at a single data center on a causally consistent snapshot. After this it immediately commits, and its updates are replicated to all other data centers in the background. This minimizes the latency of causal transactions and makes them highly available, i.e., they can be executed even when the network connections between data centers fail.

As is common in causally consistent data stores [3, 23, 43], to ensure that causal transactions execute on consistent snapshots, a data center exposes a remote transaction to clients only after exposing all its dependencies. Then to satisfy the Eventual Visibility property under failures, a data center receiving a remote causal transaction may need to forward it to other data centers, as in reliable broadcast [11] and anti-entropy protocols for replica reconciliation [54].

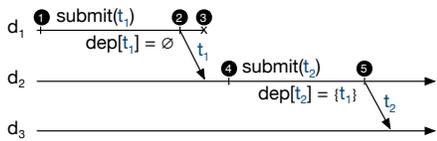


Figure 1: Why UNISTORE may need to forward remote causal transactions.

Figure 1 depicts a scenario that demonstrates how Eventual Visibility could be violated in the absence of this mechanism. Let t_1 be a causal transaction submitted at a data center d_1 (event 1). Assume that d_1 replicates t_1 to a correct data center d_2 (event 2) and then fails (event 3), so that t_1 does not get replicated anywhere else. Let t_2 be a transaction submitted at d_2 after t_1 becomes visible there, so that t_2 depends on t_1 (event 4). Transaction t_2 will eventually be replicated to all correct data centers (event 5). But it will never be exposed at any of them, because its dependency t_1 is missing. If data centers can forward remote causal transactions, then d_2 can eventually replicate t_1 to all correct data centers, preventing this problem.

On-demand strong consistency. UNISTORE uses optimistic concurrency control for strong transactions: they are first executed speculatively and the results are then *certified* to determine whether the transaction can commit, or must abort due to a conflict with a concurrent strong transaction [70]. Certifying a strong transaction requires synchronization between the replicas of partitions it accessed, located in different data centers. UNISTORE implements this using an existing fault-tolerant protocol that combines two-phase commit and Paxos [19] while minimizing commit latency. However, just using such a protocol is not enough to make the overall system fault tolerant: for this, before a strong transaction commits, all its causal dependencies must be uniform in the following sense.

DEFINITION 1. A transaction is *uniform* if both the transaction and its causal dependencies are guaranteed to be eventually replicated at all correct data centers.

This adapts the classical notion of uniformity in distributed computing to causal consistency [16]. UNISTORE considers a transaction to be uniform once it is visible at $f + 1$ data centers, because at least one of these must be correct, and data centers can forward causal transactions to others.

The following scenario, depicted in Figure 2, demonstrates why committing a strong transaction before its dependencies become uniform can compromise the liveness of the system. Assume that a causal transaction t_1 and a strong transaction t_2 are submitted at a data center d_1 in such a way that t_1 becomes a dependency of t_2 (events 1 and 2). Assume also that t_2 is certified, committed and delivered to all relevant replicas (events 3 and 4) before t_1 is replicated to any data center, and thus before it is uniform. Now if d_1 fails before

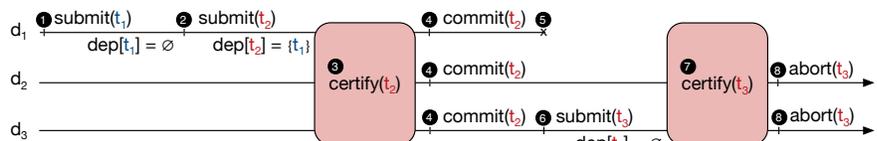


Figure 2: Why UNISTORE needs to ensure that the dependencies of a strong transaction are uniform before committing it.

replicating t_1 (event 5), no remote data center will be able to expose t_2 , because its dependency t_1 is missing. This violates the Eventual Visibility property, and even worse, no strong transaction conflicting with t_2 can commit from now on. For instance, let t_3 be such a transaction, submitted at d_3 (event 6). Because d_3 cannot expose t_2 , transaction t_3 executes on a snapshot excluding t_2 . Hence, t_3 will abort during certification (events 7 and 8): committing it would violate the Conflict Ordering property, since transactions t_2 and t_3 conflict, but neither of them is visible to the other. Ensuring that t_1 is uniform before committing t_2 prevents this problem, because it guarantees that t_1 will eventually be replicated at d_3 . After this t_2 will be exposed to conflicting transactions at this data center, which will allow them to commit.

Minimizing the latency of strong transactions. Ensuring that all the causal dependencies of a strong transaction are uniform before committing it may significantly increase its latency, since this requires additional communication between data centers. UNISTORE mitigates this problem by executing causal transactions on a snapshot that is slightly in the past, which is allowed by causal consistency. Namely, UNISTORE makes a remote causal transaction visible to the clients only after it is uniform. This minimizes the latency of a strong transaction, since to commit it only needs to wait for causal transactions originating at the local data center to become uniform. We cannot delay the visibility of the latter transactions due to the need to guarantee *read your writes* to local clients.

On-demand durability of causal transactions. Client applications interacting with the external world require hard durability guarantees: e.g., a banking application has to ensure that a withdrawal is durably recorded before authorizing the operation. UNISTORE guarantees that, once a strong transaction commits, the transaction and its dependencies are durable. However, UNISTORE returns from a causal transaction before it is replicated, and thus the transaction may be lost if its origin data center fails. Ensuring the durability of every single causal transaction would require synchronization between data centers on its critical path, defeating the benefits of causal consistency. Instead, UNISTORE reuses the mechanism for tracking uniformity to let the clients pay the cost of durability only when necessary. Even though UNISTORE replicates causal transactions asynchronously, it allows clients to execute a *uniform barrier*, which ensures that the transactions they have observed so far are uniform, and thus durable.

Client migration. Clients may need to migrate between data centers, e.g., because of roaming or for load balancing. UNISTORE also uses the uniformity mechanism to preserve session guarantees during migration. A client wishing to migrate from its local data center d to another data center i first invokes a uniform barrier at d . This guarantees that the transactions the client has observed or issued at d are durable and will eventually become visible at i , even if d fails. The client then makes an *attach* call at the destination data center i that waits until i stores all the above transactions. After this, the client can operate at i knowing that the state of the data center is consistent with the client's previous actions.

Currently UNISTORE does not support consistent client migration in response to a data center failure: if the data center a client is connected to fails, the client will have to restart its session when connecting to a different data center. As shown in [72], this limitation can be lifted without defeating the benefits of causal consistency. We leave integrating the corresponding mechanisms into UNISTORE for future work.

5 Fault-Tolerant Causal Consistency Protocol

We first describe the UNISTORE protocol for the case when all transactions are causal. We give its pseudocode in Algorithms 1 and 2; for now the reader should ignore highlighted lines, which are needed for strong transactions. For simplicity, we assume that each handler in the algorithms executes atomically (although our implementation is parallelized). We reference pseudocode lines using the format `algorithm#:line#`.

5.1 Metadata

Most metadata in our protocol are represented by vectors with an entry per each data center, where each entry stores a scalar timestamp. However, different pieces of metadata use the vectors in different ways, which we now describe.

Tracking causality. The first use of the vectors is as vector clocks [27, 47], to track causality between transactions. Given vectors V_1 and V_2 , we write $V_1 < V_2$ if each entry of V_1 is no greater than the corresponding entry of V_2 , and at least one is strictly smaller. Each update transaction is tagged with a *commit vector* $commitVec$. The order on these vectors is consistent with the causal order \prec from §3: if $commitVec_1$ and $commitVec_2$ are the commit vectors of two update transactions t_1 and t_2 such that $t_1 \prec t_2$, then $commitVec_1 < commitVec_2$. For a transaction originating at a data center d with a commit vector $commitVec$, we call $commitVec[d]$ its *local timestamp*.

Each replica p_d^m maintains a log `opLog[k]` of update operations performed on each data item k stored at the replica. Each log entry stores, together with the operation, the commit vector of the transaction that performed it. This allows reconstructing different versions of a data item from its log.

Representing causally consistent snapshots. The protocol also uses a vector to represent a snapshot of the data store

on which a transaction operates: a snapshot vector V represents all transactions with a commit vector $\leq V$. This snapshot is causally consistent. Indeed, consider a transaction t_1 included into it, i.e., $commitVec_1 \leq V$. Since any causal dependency t_0 of t_1 is such that $commitVec_0 < commitVec_1$, we have $commitVec_0 < V$, so that t_0 is also included into the snapshot. A client also maintains a vector `pastVec` that represents its *causal past*: a causally consistent snapshot including the update transactions the client has previously observed.

Tracking what is replicated where. Each replica p_d^m maintains three vectors that are used to compute which transactions are uniform. These respectively track the sets of transactions replicated at p_d^m , the local data center d , and $f + 1$ data centers. Each of these vectors V represents the set of update transactions originating at a data center i with a local timestamp $\leq V[i]$. Note that this set may not form a causally consistent snapshot. The first vector maintained by p_d^m is `knownVec`. For each data center i , it defines the prefix of update transactions originating at i (in the order of local timestamps) that p_d^m knows about.

PROPERTY 1. For each data center i , the replica p_d^m stores the updates to partition m by all transactions originating at i with local timestamps $\leq \text{knownVec}[i]$.

Our protocol ensures that $\text{knownVec}[d] \leq \text{clock}$ at any replica in data center d . The vector `knownVec` at p_d^m records whether the updates to partition m by a given transaction are stored at this replica. In contrast, the next vector `stableVec` records whether the updates to *all* partitions by a transaction are stored at the local data center d .

PROPERTY 2. For each data center i , the data center d stores the updates by all transactions originating at i with local timestamps $\leq \text{stableVec}[i]$. More precisely, we are guaranteed that $\text{knownVec}[i]$ at any replica of $d \geq \text{stableVec}[i]$ at any p_d^m .

Finally, the last vector `uniformVec` defines the set of update transactions that p_d^m knows to have been replicated at $f + 1$ data centers, including d .

PROPERTY 3. Consider `uniformVec[i]` at p_d^m . All update transactions originating at i with local timestamps $\leq \text{uniformVec}[i]$ are replicated at $f + 1$ datacenters including d . More precisely: $\text{knownVec}[i]$ at any replica of these data centers $\geq \text{uniformVec}[i]$ at p_d^m .

When `uniformVec` is reinterpreted as a causally consistent snapshot, it defines transactions that p_d^m knows to be uniform according to Definition 1:

PROPERTY 4. Consider `uniformVec` at p_d^m . All update transactions with commit vectors $\leq \text{uniformVec}$ are uniform.

Proof sketch. Consider a transaction t_1 that originates at a data center i with a commit vector $commitVec_1 \leq \text{uniformVec}$ at p_d^m . In particular, $commitVec_1[i] \leq \text{uniformVec}[i]$, and by Property 3, t_1 is replicated at $f + 1$ data centers. We as-

sume at most f failures. Then the transaction forwarding mechanism of our protocol (§4) guarantees that t_1 will eventually be replicated at all correct data centers. Consider now any causal dependency t_2 of t_1 with a commit vector $commitVec_2$. Since commit vectors are consistent with causality, $commitVec_2 < commitVec_1 \leq uniformVec$. Then as above, we can again establish that t_2 will be replicated at all correct data centers, as required by Definition 1. \square

5.2 Causal Transaction Execution

Starting a transaction. A client can submit a transaction to any replica in its local data center by calling `START_TX(V)`, where V is the client’s causal past `pastVec` (line 1:1, for brevity, we omit the pseudocode of the client). A replica p_d^m receiving such a request acts as the transaction *coordinator*. It generates a unique transaction identifier tid , computes a snapshot `snapVec[tid]` on which the transaction will execute, and returns tid to the client (we explain lines 1:2-3 and similar ones later). The snapshot is computed by combining uniform transactions from `uniformVec` (line 1:5) with the transactions from the client’s causal past originating at d (line 1:6). The former is crucial to minimize the latency of strong transactions (§4), while the latter ensures *read your writes*.

Transaction execution. The client proceeds to execute the transaction tid by issuing a sequence of operations at its coordinator via `DO_OP` (line 1:9). When the coordinator receives an operation op on a data item k , it sends a `GET_VERSION` message with the transaction’s snapshot `snapVec[tid]` to the local replica responsible for k (line 1:11). Upon receiving the message (line 1:18), the replica first ensures that it is as up-to-date as required by the snapshot (line 1:21). It then computes the latest version of k within the snapshot by applying the operations from `opLog[k]` by all transactions with commit vectors $\leq snapVec[tid]$. The result is sent to the coordinator in a `VERSION` message. After receiving it (line 1:12), the coordinator further applies the operations on k previously executed by the transaction, which are stored in a buffer `wbuff[tid]`; this ensures *read your writes* within the transaction. If the operation is an update, the coordinator then appends it to `wbuff[tid]`. Finally, the coordinator executes the desired operation op and forwards its return value to the client.

Commit. A client commits a causal transaction by calling `COMMIT_CAUSAL` (line 1:26). This returns immediately if the transaction is read-only, since it already read a consistent snapshot (line 1:28). To commit an update transaction, `UNISTORE` uses a variant of two-phase commit protocol (recall that for simplicity we only consider whole-data center failures, not those of individual replicas, §2). The coordinator first sends a `PREPARE` message to the replicas in the local data center storing the data items updated by the transaction (line 1:29). The message to each replica contains the part of the write buffer relevant to that replica. When a replica receives the message (line 1:36), it computes the transaction’s

Algorithm 1 Transaction execution at p_d^m .

```

1: function START_TX( $V$ )
2:   for  $i \in \mathcal{D} \setminus \{d\}$  do
3:      $uniformVec[i] \leftarrow \max\{V[i], uniformVec[i]\}$ 
4:    $var\ tid \leftarrow generate\_tid()$ 
5:    $snapVec[tid] \leftarrow uniformVec$ 
6:    $snapVec[tid][d] \leftarrow \max\{V[d], uniformVec[d]\}$ 
7:    $snapVec[tid][strong] \leftarrow \max\{V[strong], stableVec[strong]\}$ 
8:   return  $tid$ 

9: function DO_OP( $tid, k, op$ )
10:   $var\ l \leftarrow partition(k)$ 
11:  send GET_VERSION( $snapVec[tid], k$ ) to  $p_d^l$ 
12:  wait receive VERSION( $state$ ) from  $p_d^l$ 
13:  for all  $\langle k, op' \rangle \in wbuff[tid][l]$  do  $state \leftarrow apply(op', state)$ 
14:   $rset[tid] \leftarrow rset[tid] \cup \{\langle k, op \rangle\}$ 
15:  if  $op$  is an update then
16:     $wbuff[tid][l] \leftarrow wbuff[tid][l] \cdot \langle k, op \rangle$ 
17:  return  $retval(op, state)$ 

18: when received GET_VERSION( $snapVec, k$ ) from  $p$ 
19:  for  $i \in \mathcal{D} \setminus \{d\}$  do
20:     $uniformVec[i] \leftarrow \max\{snapVec[i], uniformVec[i]\}$ 
21:  wait until  $knownVec[d] \geq snapVec[d] \wedge$ 
     $knownVec[strong] \geq snapVec[strong]$ 
22:   $var\ state \leftarrow \perp$ 
23:  for all  $\langle op', commitVec \rangle \in opLog[k].commitVec \leq snapVec$  do
24:     $state \leftarrow apply(op', state)$ 
25:  send VERSION( $state$ ) to  $p$ 

26: function COMMIT_CAUSAL( $tid$ )
27:   $var\ L \leftarrow \{l \mid wbuff[tid][l] \neq \emptyset\}$ 
28:  if  $L = \emptyset$  then return  $snapVec[tid]$ 
29:  send PREPARE( $tid, wbuff[tid][l], snapVec[tid]$ ) to  $p_d^l, l \in L$ 
30:   $var\ commitVec \leftarrow snapVec[tid]$ 
31:  for all  $l \in L$  do
32:    wait receive PREPARE_ACK( $tid, ts$ ) from  $p_d^l$ 
33:     $commitVec[d] \leftarrow \max\{commitVec[d], ts\}$ 
34:  send COMMIT( $tid, commitVec$ ) to  $p_d^l, l \in L$ 
35:  return  $commitVec$ 

36: when received PREPARE( $tid, wbuff, snapVec$ ) from  $p$ 
37:  for  $i \in \mathcal{D} \setminus \{d\}$  do
38:     $uniformVec[i] \leftarrow \max\{snapVec[i], uniformVec[i]\}$ 
39:   $var\ ts \leftarrow clock$ 
40:   $preparedCausal \leftarrow preparedCausal \cup \{\langle tid, wbuff, ts \rangle\}$ 
41:  send PREPARE_ACK( $tid, ts$ ) to  $p$ 

42: when received COMMIT( $tid, commitVec$ )
43:  wait until  $clock \geq commitVec[d]$ 
44:   $\langle tid, wbuff, \_ \rangle \leftarrow find(tid, preparedCausal)$ 
45:   $preparedCausal \leftarrow preparedCausal \setminus \{\langle tid, \_, \_ \rangle\}$ 
46:  for all  $\langle k, op \rangle \in wbuff$  do
47:     $opLog[k] \leftarrow opLog[k] \cdot \langle op, commitVec \rangle$ 
48:   $committedCausal[d] \leftarrow committedCausal[d] \cup$ 
     $\{\langle tid, wbuff, commitVec \rangle\}$ 

49: function UNIFORM_BARRIER( $V$ )
50:  wait until  $uniformVec[d] \geq V[d]$ 

51: function ATTACH( $V$ )
52:  wait until  $\forall i \in \mathcal{D} \setminus \{d\}. uniformVec[i] \geq V[i]$ 

```

prepare time ts from its local clock and adds the transaction to preparedCausal, which stores the set of causal transactions that are prepared to commit at the replica. The replica then returns ts to the coordinator in a PREPARE_ACK message.

When the coordinator receives replies from all replicas updated by the transaction, it computes the transaction's commit vector $commitVec$: it sets the local timestamp $commitVec[d]$ to the maximum among the prepare times proposed by the replicas (line 1:33), and it copies the other entries of $commitVec$ from the snapshot vector $snapVec[tid]$ (line 1:30). The latter reflects the fact that the transactions in the snapshot become causal dependencies of tid .

After computing $commitVec$, the coordinator sends it in a COMMIT message to the relevant replicas at the local data center (line 1:34) and returns it to the client (line 1:35). The client then sets its causal past pastVec to the commit vector. When a replica receives the COMMIT message (line 1:42), it removes the transaction from preparedCausal, adds the transaction's updates to opLog, and adds the transaction to a set committedCausal[d], which stores transactions waiting to be replicated to sibling replicas at other data centers.

5.3 Transaction Replication

Each replica p_d^m periodically replicates locally committed update transactions to sibling replicas in other data centers by executing PROPAGATE_LOCAL_TXS (line 2:1). Transactions are replicated in the order of their local timestamps. The prefix of transactions that is ready to be replicated is determined by knownVec[d]: according to Property 1, p_d^m stores updates to m by all transactions originating at d with local timestamps \leq knownVec[d]. Thus, the replica first updates knownVec[d] while preserving Property 1.

There are two cases of this update. If the replica does not have any prepared transactions (preparedCausal = \emptyset), it sets knownVec[d] to the current value of the clock (line 2:2). This preserves Property 1 because in this case a new transaction originating at d and updating m will get a prepare time at m higher than the current clock (line 1:39), and thus also a higher local timestamp (line 1:33). If the replica has some prepared transactions, then they may end up getting local timestamps lower than the current clock. In this case, the replica sets knownVec[d] to just below the smallest prepared time (line 2:3). This preserves Property 1 because: (i) currently prepared transactions will get a local timestamp no lower than their prepare time; and (ii) as we argued above, new transactions will get a prepare time higher than the current clock and, hence, than the smallest prepare time.

After updating knownVec[d], the replica sends a REPLICATE message to its siblings with the transactions in committedCausal[d] such that $commitVec[d] \leq$ knownVec[d], and then removes them from committedCausal[d]. In other words, the replica sends all transactions from the prefix determined by knownVec[d] that it has not yet replicated.

When a replica p_d^m receives a REPLICATE message with

Algorithm 2 Transaction replication at p_d^m .

```

1: function PROPAGATE_LOCAL_TXS()      ▷ Run periodically
2:   if preparedCausal =  $\emptyset$  then knownVec[ $d$ ]  $\leftarrow$  clock
3:   else knownVec[ $d$ ]  $\leftarrow$  min{ $ts \mid \langle \_, \_, ts \rangle \in$  preparedCausal} - 1
4:   var txs  $\leftarrow$  { $\langle \_, \_, commitVec \rangle \in$  committedCausal[ $d$ ] |
                                      $commitVec[d] \leq$  knownVec[ $d$ ]}
5:   if txs  $\neq \emptyset$  then
6:     send REPLICATE( $d$ , txs) to  $p_i^m, i \in \mathcal{D} \setminus \{d\}$ 
7:     committedCausal[ $d$ ]  $\leftarrow$  committedCausal[ $d$ ] \ txs
8:   else send HEARTBEAT( $d$ , knownVec[ $d$ ]) to  $p_i^m, i \in \mathcal{D} \setminus \{d\}$ 
9:   when received REPLICATE( $i$ , txs)
10:  for all  $\langle tid, wbuff, commitVec \rangle \in$  txs in  $commitVec[i]$  order do
11:    if  $commitVec[i] >$  knownVec[ $i$ ] then
12:      for all  $\langle k, op \rangle \in$  wbuff do
13:        opLog[ $k$ ]  $\leftarrow$  opLog[ $k$ ]  $\cdot \langle op, commitVec \rangle$ 
14:        committedCausal[ $i$ ]  $\leftarrow$  committedCausal[ $i$ ]  $\cup$ 
                                     { $\langle tid, wbuff, commitVec \rangle$ }
15:        knownVec[ $i$ ]  $\leftarrow$   $commitVec[i]$ 
16:  when received HEARTBEAT( $i$ , ts)
17:  pre:  $ts >$  knownVec[ $i$ ]
18:  knownVec[ $i$ ]  $\leftarrow$  ts
19:  function FORWARD_REMOTE_TXS( $i$ ,  $j$ )
20:  var txs  $\leftarrow$  { $\langle \_, \_, commitVec \rangle \in$  committedCausal[ $j$ ] |
                                      $commitVec[j] >$  globalMatrix[ $i$ ][ $j$ ]}
21:  if txs  $\neq \emptyset$  then send REPLICATE( $j$ , txs) to  $p_i^m$ 
22:  else send HEARTBEAT( $j$ , knownVec[ $j$ ]) to  $p_i^m$ 
23:  function BROADCAST_VECS()          ▷ Run periodically
24:  send KNOWNVEC_LOCAL( $m$ , knownVec) to  $p_l^d, l \in \mathcal{P}$ 
25:  send STABLEVEC( $d$ , stableVec) to  $p_i^m, i \in \mathcal{D}$ 
26:  send KNOWNVEC_GLOBAL( $d$ , knownVec) to  $p_i^m, i \in \mathcal{D}$ 
27:  when received KNOWNVEC_LOCAL( $l$ , knownVec)
28:  localMatrix[ $l$ ]  $\leftarrow$  knownVec
29:  for  $i \in \mathcal{D}$  do stableVec[ $i$ ]  $\leftarrow$  min{localMatrix[ $n$ ][ $i$ ] |  $n \in \mathcal{P}$ }
30:  stableVec[strong]  $\leftarrow$  min{localMatrix[ $n$ ][strong] |  $n \in \mathcal{P}$ }
31:  when received STABLEVEC( $i$ , stableVec)
32:  stableMatrix[ $i$ ]  $\leftarrow$  stableVec
33:   $G \leftarrow$  all groups with  $f + 1$  replicas that include  $p_d^m$ 
34:  for  $j \in \mathcal{D}$  do
35:    var  $ts \leftarrow$  max{min{stableMatrix[ $h$ ][ $j$ ] |  $h \in g$ } |  $g \in G$ }
36:    uniformVec[ $j$ ]  $\leftarrow$  max{uniformVec[ $j$ ],  $ts$ }
37:  when received KNOWNVEC_GLOBAL( $l$ , knownVec)
38:  globalMatrix[ $l$ ]  $\leftarrow$  knownVec

```

a set of transactions txs originating at a sibling replica p_i^m (line 2:9), it iterates over txs in $commitVec[i]$ order. For each new transaction in txs with commit vector $commitVec$, the replica adds the transaction's operations to its log and sets knownVec[i] = $commitVec[i]$. Since communication channels are FIFO, p_d^m processes all transactions from p_i^m in their local timestamp order. Hence, the above update to knownVec[i] preserves Property 1: p_d^m stores updates originating at p_i^m by all transactions with $commitVec[i] \leq$ knownVec[i]. Finally, the replica adds the transactions to committedCausal[i], which is used to implement transaction forwarding (§4). Due to

the forwarding, p_d^m may receive the same transaction from different data centers. Thus, when processing transactions in the REPLICATE message, it checks for duplicates (line 2:11).

5.4 Advancing the Uniform Snapshot

Replicas run a background protocol that refreshes the information about uniform transactions. This proceeds in two stages. First, a replica keeps track of which transactions have been replicated at the replicas of other partitions in the same data center. To this end, replicas in the same data center periodically exchange KNOWNVEC_LOCAL messages with their knownVec vectors, which they store in a matrix localMatrix (lines 2:24 and 2:27); in our implementation this is done via a dissemination tree. This matrix is then used to compute the vector stableVec, which represents the set of transactions that have been fully replicated at the local data center as per Property 2. To ensure this, a replica computes an entry $\text{stableVec}[i]$ as the minimum of $\text{knownVec}[i]$ it received from the replicas of other partitions in the same data center (line 2:29).

In the second stage of the background protocol, sibling replicas periodically exchange STABLEVEC messages with their stableVec vectors, which they store in a matrix stableMatrix (lines 2:25 and 2:31). This matrix is then used by a replica to compute uniformVec, which characterizes the update transactions that are replicated at $f + 1$ data centers as per Property 3. To this end, a replica first enumerates all groups G of $f + 1$ data centers that include its local data center (line 2:33). For each data center j the replica performs the following computation. First, for each group $g \in G$, it computes the minimum j -th entry in the stable vectors of all data centers $h \in g$: $\min\{\text{stableMatrix}[h][j] \mid h \in g\}$. By Property 2 all update transactions originating at j with local timestamp \leq the minimum have been replicated at all data centers in g . The replica then sets $\text{uniformVec}[j]$ to the maximum of the resulting values computed for all groups $g \in G$, to cover transactions that are replicated at any such group. According to Property 4, the transactions with commit vectors \leq uniformVec are uniform, and now become visible to transactions coordinated by p_d^m (§5.2).

Replicas also update uniformVec in lines 1:2-3, 1:19-20 and 1:37-38 by incorporating $\text{snapVec}[i]$ for remote data centers i . This is safe because a transaction executes on a snapshot that only includes uniform remote transactions.

Finally, if a replica does not receive new transactions for a long time, it sends the value of its $\text{knownVec}[d]$ as a heartbeat (lines 2:8 and 2:16). This allows advancing stableVec and uniformVec even under skewed load distributions.

5.5 Transaction Forwarding

As we explained in §4, to guarantee that a transaction originating at a correct data center eventually becomes exposed at all correct data centers despite failures (Eventual Visibility), replicas may have to forward remote update transactions. To determine which transactions to forward, each replica keeps track

of the update transactions that have been replicated at sibling replicas in other data centers. To this end, sibling replicas periodically exchange KNOWNVEC_GLOBAL messages with their knownVec vectors, which they store in a matrix globalMatrix (lines 2:26 and 2:37). Thus, p_i^m has received all update transactions from p_j^m with $\text{commitVec}[j] \leq \text{globalMatrix}[i][j]$.

A replica p_d^m only forwards transactions when it suspects that a data center j may have failed before replicating all the update transactions originating at it to a data center i (this information is provided by a separate module). In this case, p_d^m executes FORWARD_REMOTE_TXS(i, j) (line 2:19). The function forwards the set of transactions txs received from p_j^m that have not been replicated at p_i^m according to $\text{globalMatrix}[i][j]$. For example, in Figure 1, UNISTORE will eventually invoke FORWARD_REMOTE_TXS(d_1, d_3) at replicas in d_2 to forward t_1 . The replica p_d^m sends the transactions in txs to p_i^m in a REPLICATE message. If there are no update transactions to forward, p_d^m sends a heartbeat to p_i^m with $\text{knownVec}[j]$.

UNISTORE periodically deletes from committedCausal transactions that have been replicated at every data center (omitted from the pseudocode for brevity).

5.6 On-Demand Durability and Client Migration

A client may wish to ensure that the transactions it has observed so far are durable. To this end, the client can call UNIFORM_BARRIER(V) at any replica in its local data center d , where V is the client's causal past pastVec (line 1:49). The replica returns to the client only when all the transactions from pastVec that originate at d are uniform, and thus durable. Then the same holds for all transactions from pastVec , because the protocol only exposes remote transactions to clients when they are already uniform (§5.2).

A client wishing to migrate from its local data center d to another data center i first calls UNIFORM_BARRIER(V) at any replica in d with $V = \text{pastVec}$, to ensure that the transactions the client has observed or issued at d will eventually become visible at i . The client then calls ATTACH(V) at any replica in i (line 1:51). The replica returns when its uniformVec includes all remote transactions from V (line 1:52). The client can then be sure that its transactions at i will operate on snapshots including all the transactions it has observed before.

6 Adding Strong Transactions

We now describe the full UNISTORE protocol with both causal and strong transactions. It is obtained by adding the highlighted lines to Algorithms 1-2 and a new Algorithm 3.

6.1 Metadata

The Conflict Ordering property of our consistency model requires any two conflicting strong transactions to be related one way or another by the causal order \prec (§3). To ensure this, the protocol assigns to each strong transaction a scalar

strong timestamp, analogous to those used in optimistic concurrency control for serializability [70]. Several vectors used as metadata in the causal consistency protocol (§5.1) are then extended with an extra *strong* entry.

First, we extend commit vectors and those representing causally consistent snapshots. Commit vectors are compared using the previous order $<$, but considering all entries; as before, this order is consistent with the causal order \prec . Furthermore, conflicting strong transactions are causally ordered according to their strong timestamps.

PROPERTY 5. For any conflicting strong transactions t_1 and t_2 with commit vectors $commitVec_1$ and $commitVec_2$, we have: $t_1 \prec t_2 \iff commitVec_1[strong] < commitVec_2[strong]$.

A consistent snapshot vector V now defines the set of transactions with a commit vector $\leq V$, according to the new $<$. The vectors $knownVec$ and $stableVec$ maintained by a replica p_d^m are also extended with a *strong* entry. The entries $knownVec[strong]$ and $stableVec[strong]$ define the prefix of strong transactions that have been replicated at p_d^m and the local data center d , respectively:

PROPERTY 6. Replica p_d^m stores the updates to m by all strong transactions with $commitVec[strong] \leq knownVec[strong]$.

PROPERTY 7. Data center d stores the updates by all strong transactions with $commitVec[strong] \leq stableVec[strong]$.

To ensure Property 7, the *strong* entry of $stableVec$ is updated at line 2:30 similarly to its other entries. We do not extend $uniformVec$, because our commit protocol for strong transactions automatically guarantees their uniformity.

6.2 Transaction Execution

UNISTORE uses optimistic concurrency control for strong transactions, with the same protocol for executing causal and speculatively executing strong transactions. To this end, Algorithm 1 is modified as follows. First, the computation of the snapshot vector $snapVec[tid]$ is extended to compute the *strong* entry (line 1:7), which is now taken into account when checking that a replica state is up to date (line 1:21). The *strong* entry of the snapshot vector is computed so as to include all strong transactions known to be fully replicated in the local data center, as defined by $stableVec[strong]$. To ensure *read your writes*, the snapshot additionally includes strong transactions from the client’s causal past, as defined by $V[strong]$. Finally, the coordinator of a transaction now maintains not only its write set, but also its read set $rset$ that records all operations by the transaction, including read-only ones (line 1:14). The latter is used to certify strong transactions.

After speculatively executing a strong transaction, the client tries to commit it by calling `COMMIT_STRONG` at its coordinator (line 3:1). The coordinator first waits until the snapshot on which the transaction operated becomes uniform by calling `UNIFORM_BARRIER` (line 3:2): as we argued in §4, this is crucial for liveness. The coordinator next submits the trans-

Algorithm 3 Committing strong transactions at p_d^m .

```

1: function COMMIT_STRONG(tid)
2:   UNIFORM_BARRIER(snapVec[tid])
3:   return CERTIFY(tid, wbuff[tid], rset[tid], snapVec[tid])

4: upon DELIVER_UPDATES(W)
5:   for all  $\langle wbuff, commitVec \rangle \in W$  in  $commitVec[strong]$  order do
6:     for all  $\langle k, op \rangle \in wbuff$  do
7:       opLog[k]  $\leftarrow$  opLog[k]  $\cdot \langle op, commitVec \rangle$ 
8:       knownVec[strong]  $\leftarrow$  commitVec[strong]

9: function HEARTBEAT_STRONG() ▷ Run periodically
10:  return CERTIFY( $\perp, \emptyset, \emptyset, \vec{0}$ )

```

action to a *certification service*, which determines whether the transaction commits or aborts (line 3:3, see §6.3). In the former case, the service also determines its commit vector, which the coordinator returns to the client. If the transaction commits, the client sets its causal past $pastVec$ to the commit vector; otherwise, it re-executes the transaction.

The certification service also notifies replicas in all data centers about updates by strong transactions affecting them via `DELIVER_UPDATES` upcalls, invoked in an order consistent with strong timestamps of the transactions (line 3:4). A replica receiving an upcall adds the new operations to its log and refreshes $knownVec[strong]$ to preserve Property 6.

Finally, a replica p_d^m that has not seen any strong transactions updating its partition m for a long time submits a dummy strong transaction that acts as a heartbeat (line 3:9). Similarly to heartbeats for causal transactions, this allows coping with skewed load distributions.

6.3 Certification Service

We implement the certification service using an existing fault-tolerant protocol from [19], with transaction commit vectors computed using the techniques from [30]. The protocol integrates two-phase commit across partitions accessed by the transaction and Paxos among the replicas of each partition. It furthermore uses white-box optimizations between the two protocols to minimize the commit latency. The use of Paxos ensures that a committed strong transaction is durable and its updates will eventually be delivered at all correct data centers (line 3:4). For each partition, a single replica functions as the Paxos leader. The protocol is described and formally specified elsewhere [19], and here we discuss it only briefly. Its pseudocode and formal specification are given in [12, §A] and [12, §C], respectively.

The certification service accepts the read and write sets of a transaction and its snapshot vector (line 3:3). Even though the service is distributed, it guarantees that commit/abort decisions are computed like in a centralized database with optimistic concurrency control – in a total *certification order*. To ensure Conflict Ordering, the decisions are computed using a concurrency-control policy similar to that for serializability [70]: a transaction commits if its snapshot includes all

conflicting transactions that precede it in the certification order. The certification service also computes a commit vector for each committed transaction by copying its per-data center entries from the transaction’s snapshot vector and assigning a strong timestamp consistent with the certification order.

7 Proof of Correctness

We have rigorously proved that UNISTORE correctly implements the specification of PoR consistency for the case when the data store manages last-writer-wins registers. The proof uses the formal framework from [14, 15, 17] and establishes Properties 1-7 stated earlier. Due to space constraints, we defer the proof to [12, §D].

8 Evaluation

We have implemented UNISTORE and several other protocols (listed in the following) in the same codebase, consisting of 10.3K SLOC of Erlang. We evaluate the protocols on Amazon EC2 using m4.2xlarge VMs from 5 different regions. Each VM has 8 virtual cores and 32GB of RAM. The RTT between regions ranges from 26ms to 202ms. Unless otherwise stated, our experiments deploy 3 data centers, thus tolerating a single data center failure: Virginia (US-East), California (US-West) and Frankfurt (EU-FRA). All Paxos leaders are located in Virginia. By default we use 4 replica machines per data center. Each machine stores replicas of 8 partitions, matching the number of cores. Clients are hosted on separate machines in each data center. We run each experiment for at least 5 minutes, with the first and the last minute ignored. Replicas propagate local update transactions (line 2:1) and broadcast vectors (line 2:23) every 5ms.

8.1 Does UNISTORE combine causal and strong consistency effectively?

We start by analyzing the performance of UNISTORE using RUBiS – a popular benchmark that emulates an online auction website such as eBay [41, 42]. It defines 11 read-only transactions and 5 update transactions, e.g., selling items, bidding on items, and consulting outstanding auctions. As in previous work [42], to make the benchmark more challenging, we add an extra update transaction `closeAuction` to declare the winner of an auction. We also borrow from [42] a conflict relation between RUBiS transactions that preserves key integrity invariants in the PoR consistency model. This marks four transactions as strong (`registerUser`, `storeBuyNow`, `storeBid` and `closeAuction`) and declares three conflicts between them. For example, `storeBid`, which places a bid on a item, conflicts with `closeAuction` if both act on the same item: this is needed to preserve the invariant that the winner of an auction is the highest bidder. Our RUBiS database is configured according to the benchmark specification: it is populated with 33,000 items for sale and 1 million users; client

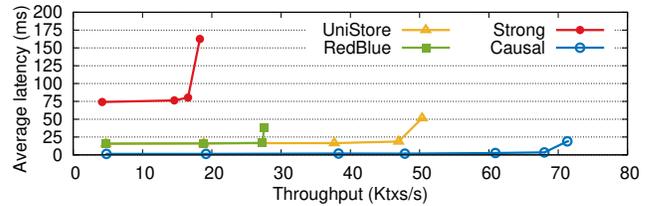


Figure 3: RUBiS benchmark: throughput vs. average latency.

think times are 500ms. We run the bidding mix workload of RUBiS with 15% of update transactions, which yields 10% of strong transactions.

We compare UNISTORE with STRONG, REDBLUE and CAUSAL. STRONG implements serializability [70] as a special case of UNISTORE where all transactions are strong and all pairs of operations conflict. REDBLUE implements red-blue consistency [41], which like PoR, combines causal and strong consistency. However, it declares conflicts between all strong transactions. REDBLUE certifies strong transactions at a centralized replicated service, with a replica at each data center. CAUSAL implements causal consistency as a special case of UNISTORE where all transactions are causal. It cannot preserve the integrity invariants of RUBiS, but gives an upper bound on the expected performance.

Throughput and average latency. Figure 3 evaluates average transaction latency and throughput. As the figure shows, UNISTORE exhibits a high throughput: 72% and 183% higher than REDBLUE and STRONG respectively at their saturation point. This is expected, as UNISTORE implements the consistency model that enables the most concurrency. STRONG classifies all transactions as strong. This impacts performance because executing a strong transaction is significantly more expensive than executing a causal one. REDBLUE uses a centralized certification service that saturates before the UNISTORE’s distributed service, creating a bottleneck. UNISTORE exhibits an average latency of 16.5ms, lower than 80.4ms of STRONG. The latency of REDBLUE is comparable to that of UNISTORE. This is because both systems mark the same set of transactions as strong. Still, REDBLUE declares conflicts between all strong transactions and thus aborts more transactions than UNISTORE: 0.12% vs 0.027%. The clients whose transactions abort have to retry them, thus increasing latency. Since the abort rate remains low in both cases, the difference in latency is negligible in our experiment. We expect a more significant difference in workloads with higher contention. Finally, in comparison to CAUSAL, UNISTORE penalizes throughput by 45%. This is the unavoidable price to pay to preserve application-specific invariants.

Latency of each transaction type. In UNISTORE, the latency of strong transactions is dominated by the RTT between Virginia (the leader’s region) and California (Virginia’s closest data center) – 61ms. Strong transactions exhibit a latency of 73.9ms on average. The latency varies depending on the

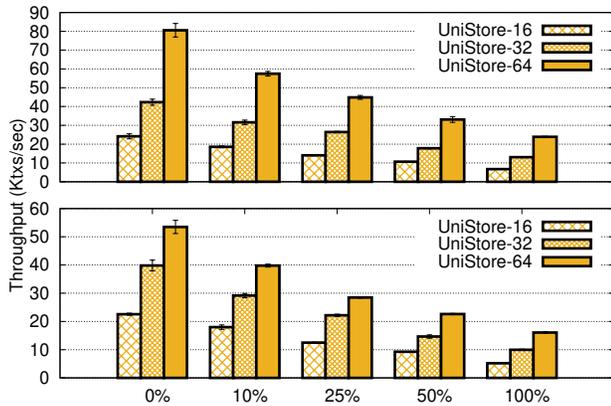


Figure 4: Scalability when varying the ratio of strong transactions with uniform data access (top) and under contention (bottom).

client’s location: from 65.4ms on average at the leader’s site to 93.2ms at the site furthest from the leader (Frankfurt). Since causal transactions do not require coordination between data centers, they exhibit a very low latency – 1.2ms on average, which is comparable to that of CAUSAL. This demonstrates that UNISTORE is able to mix causal and strong consistency effectively, as the latency of causal transactions remains low regardless of concurrently executing strong transactions.

8.2 How does UNISTORE scale with the number of machines?

We evaluate the peak throughput of UNISTORE as we increase the number of machines per each data center from 2 to 8, i.e., the number of partitions from 16 to 64. We use a microbenchmark with 100% of update transactions, where each transaction accesses three data items. We vary the ratio of strong transactions from 0% to 100% to understand their impact on scalability.

Scalability under low contention. For this set of experiments, the data items accessed by each transaction are picked uniformly at random. This yields a very low contention: e.g., with 16 partitions, the probability of two transactions accessing the same partition is 0.031. As shown by the top plot of Figure 4, UNISTORE is able to scale almost linearly even when the workload includes strong transactions: a 9.76% throughput drop compared to the optimal scalability. This is because, with uniform accesses, the task of committing transactions is balanced among partitions. Thus, when the number of partitions increases, so does the system’s capacity. The scalability is not perfect due to the cost of the background protocol that computes stableVec, which grows logarithmically with the number of partitions. The plot also shows that strong transactions are expensive: 25.72% of throughput drop on average with 10% of strong transactions. The performance is dominated by the number of strong transactions that a partition can certify per second.

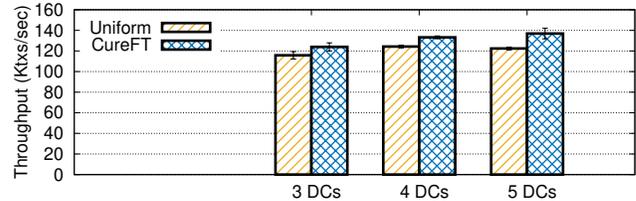


Figure 5: Throughput penalty of tracking uniformity.

Impact of contention. For this set of experiments, we set the ratio of strong transactions that access a designated partition to 20% to create contention. As shown by the bottom plot of Figure 4, UNISTORE is still able to scale fairly well under contention. But, as expected, contention has an impact on scalability: a 17.15% throughput drop from the optimal scalability compared to the 9.76% throughput drop in the experiments without contention.

8.3 What is the cost of uniformity?

We compare CUREFT to UNIFORM. CUREFT implements Cure [3], a causally consistent data store, and makes it fault tolerant by adding transaction forwarding (§4). UNIFORM is a simplified version of UNISTORE that removes all the mechanisms related to strong transactions. UNIFORM tracks uniformity and makes remote transactions visible only when these are uniform; CUREFT does not. We use a microbenchmark with only causal transactions and 15% of update transactions. Each transaction accesses three data items.

Throughput penalty. Figure 5 evaluates the cost of tracking uniformity. It shows the peak throughput when the number of data centers increases from 3 to 5. We first add Ireland and then Brazil. As we do this, the throughput remains almost constant. This is because each data center stores replicas of all partitions and the computational power gained when adding a data center is offset by the cost of replicating update transactions. As the figure shows, the cost of tracking uniformity is small: a 7.97% drop on average. The gap grows as we increase the number of data centers: a 10.61% drop on average with 5 data centers. This is because, to track uniform transactions, sibling replicas exchange messages: the more data centers, the more messages exchanged. The penalty can be reduced by decreasing the frequency at which sibling replicas exchange their stableVec (line 2:25), at the expense of an extra delay in the visibility of remote transactions.

Reading from a uniform snapshot. Figure 6 evaluates the delay on the visibility of remote transactions when reading from a uniform snapshot. We deploy four data centers: Virginia, California, Frankfurt and Brazil. We set $f = 2$ to tolerate 2 data center failures (when $f = 1$, UNIFORM shows no delay). Under such a configuration, a data center makes a transaction visible when it knows that 3 data centers store the transaction and its dependencies (§5.4). The figure shows the cumulative distribution of the delay before updates from

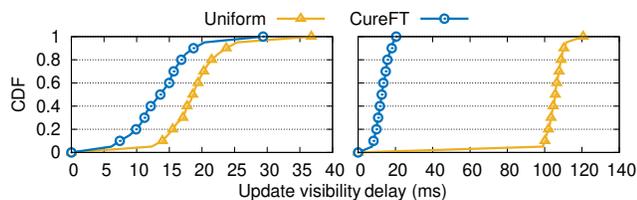


Figure 6: Left: California to Brazil (best case). Right: California to Virginia (worst case).

California are visible in Brazil and Virginia.

The extra delay at Brazil is only of 5ms at the 90th percentile. This is the best case scenario for UNIFORM because Brazil learns that Virginia stores a transaction originating at California only 2ms after receiving it. The worst case scenario for UNIFORM is when the origin and the destination data center are the closest ones. This is why the extra delay at Virginia is 92ms at the 90th percentile: Brazil learns that Frankfurt stores a transaction originating at California 88ms after receiving it. Note that when clients communicate only via the data store, the delay is unnoticeable. Even if clients communicate out of band, as the maximum extra delay is less than 100ms, it is unlikely that a client will miss an update.

9 Related Work

Systems with multiple consistency levels. A number of data stores have combined weak and strong consistency, including several commercial and academic systems that combine eventual and strong consistency [5, 6, 29, 49, 58, 65, 73]. Several academic data stores combined causal and strong consistency [9, 37, 41, 42, 59, 65]. Pileus [65] funnels all updates through a single data center. In the fault-tolerant version of lazy replication [37], causal operations require synchronization between replicas on its critical path. In both cases, causal operations are not highly available, defeating the benefits of causal consistency. Walter [59] restricts causal operations to a specific type and lacks fault tolerance due to the use of two-phase commit across data centers. The remaining works [9, 41, 42] support highly available causal operations, but are not fault tolerant. First, they do not make causal operations uniform on demand to guarantee the liveness of strong operations. Thus, they suffer from the liveness issue we explained in §4 (Figure 2). In addition, these systems do not use fault-tolerant mechanisms even for strong transactions. They guard the use of strong transactions using mechanisms similar to locks; if the lock holder fails before releasing it, no other data center can execute a strong transaction requiring the same lock. This occurs even when the service handing locks is fault-tolerant, as in [42]. Finally, the above systems either do not include mechanisms for partitioning the key space among different machines in a data center or include per-data center centralized services, which limits their scalability (§8.2).

Some group communication systems mix causal and atomic

broadcast [10, 68]. However, these systems do not provide mechanisms for maintaining transactional data consistency.

Several papers have proposed tools that use formal verification technology to ensure that consistency choices do not violate application invariants [9, 31, 34, 40, 51, 52]. Such tools can make it easier for programmers to use our system.

Causal consistency implementations. Our subprotocol for causal consistency belongs to a family of highly scalable protocols that avoid using any centralized components or dependency check messages [3, 23, 60–62]; other alternatives are less scalable [4, 8, 13, 22, 32, 43, 44, 48, 72]. While we base our causal consistency subprotocol on an existing one, Cure [3], we have extended it in nontrivial ways, by integrating mechanisms for tracking uniformity (§5.4) and for transaction forwarding (§5.5). Some of the above protocols [32, 61] use hybrid clocks instead of real time [36] to improve performance with large clock skews; this technique can also be integrated into UNISTORE.

SwiftCloud [72] implements *k-stability* [46], a notion similar to uniformity, to enable client migration. SwiftCloud relies on a single per-data center sequencer, which makes tracking *k-stability* easy, but the data store less scalable. Our protocol is more sophisticated, since we distribute the responsibility of tracking uniformity among the replicas in a data center.

Paxos variants. Several Paxos variants [25, 39, 50, 53] lower the latency by allowing commutative operations to execute at replicas in arbitrary orders. In contrast to them, UNISTORE implements PoR consistency, which allows causal transactions to execute without any synchronization at all.

10 Conclusion

This paper presented UNISTORE, the first fault-tolerant and scalable data store that combines causal and strong consistency. UNISTORE carefully integrates state-of-the-art scalable protocols and extends them in nontrivial ways. To maintain liveness despite data center failures, unlike previous work, UNISTORE commits a strong transaction only when all its causal dependencies are uniform. Our results show that UNISTORE combines causal and strong consistency effectively: 3.7× lower latency on average than a strongly consistent system with 1.2ms latency on average for causal transactions. We expect that the key ideas in UNISTORE will pave the way for practical systems that combine causal and strong consistency.

Acknowledgements. We thank our shepherd, Heming Cui, as well as Gregory Chockler, Vitor Enes, Luís Rodrigues and Marc Shapiro for comments and suggestions. This work was partially supported by an ERC Starting Grant RACCOON, the Juan de la Cierva Formación funding scheme (FJC2018-036528-I), the CCF-Tencent Open Fund (CCF-Tencent RAGR20200124) and the AWS Cloud Credit for Research program.

References

- [1] D. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2), 2012.
- [2] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Comput.*, 9(1), 1995.
- [3] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Pregoça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *International Conference on Distributed Computing Systems (ICDCS)*, 2016.
- [4] S. Almeida, J. Leitão, and L. Rodrigues. ChainReaction: A causal+ consistent datastore based on chain replication. In *European Conference on Computer Systems (EuroSys)*, 2013.
- [5] Amazon. Read consistency. <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html>, 2020.
- [6] Apache Cassandra. Read repair. https://cassandra.apache.org/doc/latest/operating/read_repair.html, 2020.
- [7] H. Attiya, F. Ellen, and A. Morrison. Limitations of highly-available eventually-consistent data stores. *IEEE Trans. Parallel Distributed Syst.*, 28(1), 2017.
- [8] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *International Conference on Management of Data (SIGMOD)*, 2013.
- [9] V. Balesgas, N. Pregoça, R. Rodrigues, S. Duarte, C. Ferreira, M. Najafzadeh, and M. Shapiro. Putting the consistency back into eventual consistency. In *European Conference on Computer Systems (EuroSys)*, 2015.
- [10] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3), 1991.
- [11] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1), 1987.
- [12] M. Bravo, A. Gotsman, B. de Régil, and H. Wei. UniStore: A fault-tolerant marriage of causal and strong consistency (extended version). *arXiv CoRR*, abs/2106.00344, 2021.
- [13] M. Bravo, L. Rodrigues, and P. Van Roy. Saturn: A distributed metadata service for causal consistency. In *European Conference on Computer Systems (EuroSys)*, 2017.
- [14] S. Burckhardt. *Principles of Eventual Consistency*. Now Publishers, 2014.
- [15] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality. In *Symposium on Principles of Programming Languages (POPL)*, 2014.
- [16] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2nd ed.)*. Springer, 2011.
- [17] A. Cerone, G. Bernardi, and A. Gotsman. A framework for transactional consistency models with atomic visibility. In *International Conference on Concurrency Theory (CONCUR)*, 2015.
- [18] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips. Giza: Erasure coding objects across global data centers. In *USENIX Annual Technical Conference (USENIX ATC)*, 2017.
- [19] G. Chockler and A. Gotsman. Multi-shot distributed transaction commit. In *Symposium on Distributed Computing (DISC)*, 2018.
- [20] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google’s Globally-Distributed Database. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.
- [21] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Voshall, and W. Vogels. Dynamo: Amazon’s highly available key-value store. In *Symposium on Operating Systems Principles (SOSP)*, 2007.
- [22] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Symposium on Cloud Computing (SoCC)*, 2013.
- [23] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Symposium on Cloud Computing (SoCC)*, 2014.
- [24] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2), 1988.

- [25] V. Enes, C. Baquero, T. F. Rezende, A. Gotsman, M. Perin, and P. Sutra. State-machine replication for planet-scale systems. In *European Conference on Computer Systems (EuroSys)*, 2020.
- [26] FaunaDB. What is FaunaDB? <https://docs.fauna.com/fauna/current/introduction.html>, 2020.
- [27] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Australian Computer Science Conference (ASCS)*, 1988.
- [28] S. Gilbert and N. Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), 2002.
- [29] Google. Balancing strong and eventual consistency with datastore. <https://cloud.google.com/datastore/docs/articles/balancing-strong-and-eventual-consistency-with-google-cloud-datastore>, 2020.
- [30] A. Gotsman, A. Lefort, and G. Chockler. White-box atomic multicast. In *International Conference on Dependable Systems and Networks (DSN)*, 2019.
- [31] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. ‘Cause I’m strong enough: reasoning about consistency choices in distributed systems. In *Symposium on Principles of Programming Languages (POPL)*, 2016.
- [32] C. Gunawardhana, M. Bravo, and L. Rodrigues. Unobtrusive deferred update stabilization for efficient geo-replication. In *USENIX Annual Technical Conference (USENIX ATC)*, 2017.
- [33] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.
- [34] G. Kaki, K. Earanky, K. C. Sivaramakrishnan, and S. Jagannathan. Safe replication through bounded concurrency verification. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018.
- [35] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *European Conference on Computer Systems (EuroSys)*, 2013.
- [36] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone. Logical physical clocks. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2014.
- [37] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4), 1992.
- [38] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998.
- [39] L. Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.
- [40] C. Li, J. Leitão, A. Clement, N. M. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *USENIX Annual Technical Conference (USENIX ATC)*, 2014.
- [41] C. Li, D. Porto, A. Clement, R. Rodrigues, N. Preguiça, and J. Gehrke. Making geo-replicated systems fast if possible, consistent when necessary. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.
- [42] C. Li, N. Preguiça, and R. Rodrigues. Fine-grained consistency for geo-replicated systems. In *USENIX Annual Technical Conference (USENIX ATC)*, 2018.
- [43] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don’t settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Symposium on Operating Systems Principles (SOSP)*, 2011.
- [44] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Conference on Networked Systems Design and Implementation (NSDI)*, 2013.
- [45] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. Technical Report TR-11-22, University of Texas at Austin, 2011.
- [46] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *ACM Trans. Comput. Syst.*, 29(4), 2011.
- [47] F. Mattern. Virtual time and global clocks in distributed systems. In *International Workshop on Parallel and Distributed Algorithms*, 1988.
- [48] S. A. Mehdi, C. Littlely, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd. I can’t believe it’s not causal! Scalable causal consistency with no slowdown cascades. In *Conference on Networked Systems Design and Implementation (NSDI)*, 2017.
- [49] Microsoft. Consistency levels in Azure Cosmos DB. <https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels>, 2020.

- [50] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Symposium on Operating Systems Principles (SOSP)*, 2013.
- [51] S. S. Nair, G. Petri, and M. Shapiro. Proving the safety of highly-available distributed objects. In *European Symposium on Programming (ESOP)*, 2020.
- [52] M. Najafzadeh, A. Gotsman, H. Yang, C. Ferreira, and M. Shapiro. The CISE tool: proving weakly-consistent applications correct. In *Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC)*, 2016.
- [53] F. Pedone and A. Schiper. Generic broadcast. In *International Symposium on Distributed Computing (DISC)*, 1999.
- [54] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Symposium on Operating Systems Principles (SOSP)*, 1997.
- [55] Redis Labs. Causal consistency in an active-active database. <https://docs.redislabs.com/latest/rs/administering/database-operations/causal-consistency-crdb/>, 2020.
- [56] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4), 1990.
- [57] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2011.
- [58] D. Shukla, S. Thota, K. Raman, M. Gajendran, A. Shah, S. Ziuzin, K. Sundaram, M. G. Guajardo, A. Wawrzyniak, S. Boshra, R. Ferreira, M. Nassar, M. Koltachev, J. Huang, S. Sengupta, J. J. Levandoski, and D. B. Lomet. Schema-agnostic indexing with azure documentdb. *Proc. VLDB Endow.*, 8(12), 2015.
- [59] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Symposium on Operating Systems Principles (SOSP)*, 2011.
- [60] K. Spirovska, D. Didona, and W. Zwaenepoel. Wren: Nonblocking reads in a partitioned transactional causally consistent data store. In *International Conference on Dependable Systems and Networks (DSN)*, 2018.
- [61] K. Spirovska, D. Didona, and W. Zwaenepoel. Paris: Causally consistent transactions with non-blocking reads and partial replication. In *International Conference on Distributed Computing Systems (ICDCS)*, 2019.
- [62] K. Spirovska, D. Didona, and W. Zwaenepoel. Optimistic causal consistency for geo-replicated key-value stores. *IEEE Transactions on Parallel and Distributed Systems*, 32(3), 2020.
- [63] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *International Conference on Management of Data (SIGMOD)*, 2020.
- [64] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *International Conference on Parallel and Distributed Information Systems (PDIS)*, 1994.
- [65] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Symposium on Operating Systems Principles (SOSP)*, 2013.
- [66] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Symposium on Operating Systems Principles (SOSP)*, 1995.
- [67] M. Tyulenev, A. Schwerin, A. Kamsky, R. Tan, A. Cabral, and J. Mulrow. Implementation of cluster-wide logical clock and causal consistency in MongoDB. In *International Conference on Management of Data (SIGMOD)*, 2019.
- [68] R. van Renesse, K. P. Birman, and S. Maffei. Horus: A flexible group communication system. *Commun. ACM*, 39(4), 1996.
- [69] W. Vogels. Eventually consistent. *CACM*, 52(1), 2009.
- [70] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., 2001.
- [71] YugabyteDB. Replication. <https://docs.yugabyte.com/latest/architecture/docdb-replication/replication/>, 2020.
- [72] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balesgas, and M. Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *International Middleware Conference (Middleware)*, 2015.
- [73] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. *ACM Trans. Comput. Syst.*, 35(4), 2018.

Optimistic Concurrency Control for Real-world Go Programs

Zhizhou Zhang¹, Milind Chabbi², Adam Welc², and Timothy Sherwood¹

¹University of California, Santa Barbara

¹{zhizhouzhang, sherwood}@cs.ucsb.edu

²Programming Systems Group, Uber Technologies

²{milind, adam.welc}@uber.com

Abstract

We present a source-to-source transformation framework, GOCC, that consumes lock-based pessimistic concurrency programs in the Go language and transforms them into optimistic concurrency programs that use Hardware Transactional Memory (HTM). The choice of the Go language is motivated by the fact that concurrency is a first-class citizen in Go, and it is widely used in Go programs. GOCC performs rich inter-procedural program analysis to detect and filter lock-protected regions and performs AST-level code transformation of the surrounding locks when profitable. Profitability is driven by both static analyses of critical sections and dynamic analysis via execution profiles. A custom HTM library, using perceptron, learns concurrency behavior and dynamically decides whether to use HTM in the rewritten lock/unlock points. Given the rich history of transactional memory research but its lack of adoption in any industrial setting, we believe this workflow, which ultimately produces source-code patches, is more apt for industry-scale adoption. Results on widely adopted Go libraries and applications demonstrate significant (up to 10×) and scalable performance gains resulting from our automated transformation while avoiding major performance regressions.

1 Introduction

Golang [46] (or simply Go) is a modern programming language that has gained significant popularity over the last decade. It is being used to write enterprise software [20] (e.g., to implement backend services) in some of the largest technology companies as well as to develop large and widely used open-source applications (e.g., Kubernetes [47]) and libraries (e.g., Tally [88]). The design of Go is inspired by C, but unlike C, it supports concurrency as the first-class language construct. Even more importantly, and unlike other popular languages with first-class concurrency support (e.g., Java), the Go language goes to great lengths to simplify concurrent programming by making concurrency easy to use (and thus frequently used) by the developers [86] — any function in Go can be

scheduled to execute concurrently with the rest of the code as a *goroutine* [46] by simply prefixing its call with the `go` keyword.

Although Go makes writing concurrent programs easier, it still requires programmers to manage interactions between concurrently executing code — this can be accomplished either via passing messages through channels [46] or explicitly synchronizing accesses to shared memory. Shared memory is used more often than message passing by Go developers, and mutual exclusion via locks [11] remains the most widely-used synchronization mechanism across several applications [86]. It is, therefore, the focus of our work.

Locks may unnecessarily serialize concurrent execution, even if the code operates on disjoint data. Our work aims to improve the performance of concurrent Go code, particularly code hiding behind needlessly held locks. Our goal is to accomplish this while retaining the correctness of concurrent execution. We utilize the concept of *transactional memory* (TM) [53] to achieve this goal. The general idea behind TM is to decide on whether two (or more) pieces of code can be executed concurrently based on whether their accesses to the underlying data are *conflicting* [54] or not, that is, if at least one of the accesses is a write. Conflict-free executions are allowed to proceed in parallel. On the other hand, upon encountering a data access conflict, execution effects of at least one piece of code have to be *rolled back* (i.e., undone), and the computation must be restarted. TM machinery, which originally started in software (STM) [43, 77, 82, 93], is now available in commodity hardware as Hardware Transactional Memory (HTM) [41, 90, 92]. However, despite almost three decades of work in this area, TM’s promise of accelerating concurrent computations for real-life software has not been quite fulfilled. We speculate that there are two reasons why this is the case.

The first reason is that TM, while being a single concept, may have different realizations in terms of algorithms and implementations (e.g., eager vs. lazy versioning [80]) and different integration strategies at the language level (e.g., API-level solutions [77] or the compiler-assisted `atomic` construct used to demarcate TM-managed concurrent code [52]), resulting in different behavior from the programmer’s perspective. Conse-

quently, attempts to introduce TM as a separate language-level mechanism lead to significant semantic dissonance with respect to existing concurrency-related mechanisms [69, 79].

The second reason is that a lot of TM (particularly STM) work was focusing on designing and implementing TM algorithms but limiting empirical evaluation to synthetic benchmarks (e.g., STMBench7 [51]) or measuring the performance of only selected concurrent data structures. Unfortunately, unlike what was expected, TM techniques did not easily generalize to real-life applications [93]. A few attempts to apply TM to production code were unsuccessful (e.g., an attempt to rewrite the Quake game server to use TM [97]).

In this work, we attempt to rectify some of these limitations and show that TM can be effective in accelerating real-life concurrent code. We focus less on the algorithmic side of TM (we use state-of-the-art off-the-shelf HTM implementation from Intel), and more on how and when to apply the TM machinery to maximize the benefit. Additionally, we replace Go locks with HTM constructs without changing the code’s behavior in any way, which allows us to completely bypass complications related to transactional memory semantics. More specifically, we employ *transactional lock elision* (TLE) [73] — a well-known technique that attempts to execute a lock-protected critical section as an atomic hardware transaction, reverting to using the lock if these attempts fail.

Figure 1 depicts our solution. At a high level, our solution starts with using static analysis to identify candidate lock-protected critical sections to be instead protected by the HTM. Then we filter out non-desirable candidates using both static analyses (e.g., to eliminate regions containing I/O operations) and dynamic analysis (to eliminate regions where the application of the HTM would not be beneficial based on profile data collected at runtime). Finally, we rewrite the code to have candidate regions use HTM constructs provided by the HTM library we developed instead of Go locks [11]. GOCC transformations are guaranteed to be safe; developer involvement is optional but highly recommended to let developers ultimately decide whether or not they want to use HTM.

In summary, this paper makes the following contributions:

1. We present the design and implementation of a framework for identifying lock-protected critical sections and select the best candidates for lock elision based on static analysis and execution profiles of Go programs.
2. We describe the source-to-source code transformation to replace mutual-exclusion locks in Go programs with HTM concurrency control constructs.
3. We introduce a library extending vendor-provided HTM primitives with intelligent features such as runtime contention management. Specifically, we devise a lightweight perceptron [59, 84] that learns whether eliding a lock via HTM at a call site [50] is beneficial at runtime.
4. We demonstrate the effectiveness of GOCC for improving performance of real-life concurrent Go code by up to 10×.

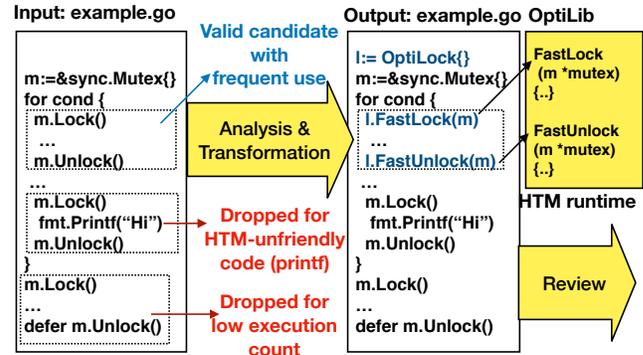


Figure 1: GOCC schematic diagram. Static analysis detects three legal lock-unlock pairs in the input file `example.go`. The top one is a valid replacement candidate. The middle one is filtered since it contains I/O operations in its critical section. The bottom one is dropped due to the infrequent use via the information provided by profiles. The transformed code calls `optiLib`, which executes the critical section via HTM. The resulting diff is given to the developer for review.

2 Challenges

Locks are widely used in the real-world Go code and a significant amount of execution time can be spent waiting to acquire them [15, 19, 27, 28, 68, 86, 89]¹. It is possible to replace a lock with a transaction that enables a critical section to be speculatively executed without actually holding the guarding lock. With the support of the HTM, such replacements can result in significant speedups. However, there are several challenges in performing these replacements correctly and robustly, and ensuring that they deliver high performance reliably.

First, automatically and accurately matching a lock with its corresponding unlock operation to precisely identify critical sections is a complex problem. Real-world programs can use locks with nesting intra- or inter-procedurally, which makes it significantly more involved. Additionally, certain lock-compatible instructions (e.g., IO and privileged instructions) will not work with HTM. A critical section including such instructions will not benefit from HTM.

Furthermore, Go provides a keyword that enables delaying lock release operation to all exit points of a function by prefixing the `Unlock()` operation with the `defer` [49] keyword². It not only complicates matching an unlock with a lock operation, it may unnecessarily lengthen a critical section, which according to a synthetic benchmark we wrote shows performance degradation. A scan of 21 million lines of industrial Go code, which includes about 8000 `Unlock()` operations, shows that about 76% are prefixed with the `defer` keyword. This indicates that handling `defer` statements is important.

Second, the Go language nuances [46] (e.g., pointer vs. value syntax, anonymous `Mutex` fields, lambda functions, etc.) make

¹ A limited study we performed in a large-scale industrial setting using thousands of different Go services showed up to 30% execution time being spent in lock-related code in certain Go programs; 5-10% was quite common.

² Any function can be deferred in Go.

it non-trivial to transform lock-based code to HTM-based code.

Third, HTM has startup and commit overheads. Even in non-concurrent code, where data-access conflicts do not happen, HTM can fail [14], and locks may outperform HTM, particularly on tiny critical sections [75].

Fourth, the critical section size can be hard to estimate in general. If we make the conservative design choice and do not replace the lock if the critical section size is unknown, we can miss the opportunity to generate significant performance improvement. Thus, we need some runtime mechanism that can handle critical sections of arbitrary sizes with low overhead.

Fifth, when HTM aborts for a genuine data-access conflict, naively falling back to using a lock can be detrimental to performance [42, 64]. Deciding when and how to retry HTM-based executions or fall back to using fine-grained locks must be handled very carefully to avoid pathologies [37, 42, 64].

Our tool, GOCC, attempts to solve the above challenges. GOCC is an end-to-end system for improving the performance of lock-based Go code using HTM. We devise a sophisticated program analysis to identify lock-protected critical sections (§ 5.2), support lock-to-HTM code transformation including non-trivial Go features (§ 5.3), and develop an efficient HTM library to handle issues manifested at runtime (§ 5.4).

3 Related Work

Herlihy and Moss proposed transactional Memory (TM) [53] in 1993 as an alternative to locks. While locks proactively prevent two or more threads from concurrently accessing shared data, TM takes the opposite approach — concurrent accesses are allowed as long as they do not conflict. A lot of work has been done around both software and hardware implementations of transactional memory [41, 43, 71, 77, 82, 90, 92], but only a few [61, 76, 93, 97] focused on evaluating the approach with real-life workloads, and none have done this for Go.

Intel’s TSX extension of x86 instructions set [5] implementing HTM is of specific interest here as it underlies parts of our implementation. It is widely available in modern Intel CPUs and offers software interfaces providing subtly different functionality. The RTM (Restricted Transactional Memory) interface allows programmers to execute arbitrary code as a hardware transaction. All operations within a transaction have atomic execution behavior — they all either appear to happen instantaneously or the entire transaction aborts and reverts the architectural state to before it was started. This can be trivially used to emulate the behavior of mutual-exclusion locks. In fact, this is precisely the kind of functionality that the HLE (Hardware Lock Elision) interface provides. However, HLE has been introduced mainly for backward compatibility with architectures that are not TSX-enabled and is not only very simplistic (e.g., with respect to contention management) but has also been shown to perform poorly compared to RTM [1]. Consequently, our solution uses the RTM interface as the low-level implementation mechanism to build a comprehensive

TM-based alternative for mutual-exclusion locks.

Lock elision, whether in software or hardware or a hybrid fashion, including gaining insights into them, has been extensively studied [22, 29, 34, 36–39, 44, 45, 57, 58, 63, 71, 71, 72, 74, 81, 91, 92, 95]. Our work uses many of those techniques; for example, the basic design of our runtime controller was inspired by Wang et al. [91]. Additional possibilities to bring more solutions from the literature to the design and implementation of both our static analysis tool and runtime controller also exist. Other attempts to use transactional memory for emulating mutual-exclusion locks exist as well [70, 96], but they have to cope with higher overheads and semantics-related complications due to using the STM, they target the Java language whose synchronization lock-like primitives (i.e., *monitors*) are easier to handle due to their lexical scoping and, most importantly, their evaluation is based exclusively on synthetic benchmarks.

4 GOCC Overview

A Go `Mutex` is a runtime object with `Lock()` and `Unlock()` operations on it. Two (or more) critical sections guarded by the same `Mutex` will not execute concurrently. When transforming locks into HTM, there are two possibilities.

1. A given `Mutex` guarding a set of critical sections is replaced with another object supporting operations analogous to `Lock()/Unlock()` but provided by the HTM. As a result, all critical sections previously guarded by the `Mutex` are now executed under HTM’s control.
2. `Lock()/Unlock()` operations of the `Mutex` are replaced with their HTM equivalents on a per critical section basis. As a result, some critical sections for a given `Mutex` are still guarded by the same `Mutex`, while the others execute under HTM’s control.

The former is doable only if it is beneficial to transform all `Lock()/Unlock()` operations using a given `Mutex`, and the `Mutex` object is defined in the code that we are rewriting. Assessing the benefit of transforming the `Mutex` object would require inspecting every critical section it protects. A “may alias” pointer analysis [55, 65] can answer such a question. The “all-or-none” coarse-granularity of this approach makes it unattractive because the imprecision of pointer analysis overapproximates the critical sections protected by a `Mutex`, disqualifying too many `Mutexes` from transformation.

This work adopts the latter approach, where we consider pairs of `Lock()/Unlock()` operations in the code for transformation, which provides fine-grained control over transformation. This approach has to handle pairing a lock with its corresponding unlock and support interoperability of HTM (where the code is transformed) with locks (where the code is not transformed). This kind of interoperability is well-studied in the literature [23, 32, 40, 41, 64] and is handled by our library.

Recall, from Figure 1, that input to GOCC is the source code for a Go package along with its execution profiles. The output

is a source code patch, where candidate `Lock()/Unlock()` operations are replaced with calls to a custom HTM library. GOCC consists of the following key components:

- Analyzer: performs static analysis on the input program and collects lock-unlock pairs for transformation (§ 5.2).
- Transformer: rewrites the program by replacing `Lock()/Unlock()` with `FastLock()/FastUnlock()`, which elide the lock using HTM (§ 5.3).
- Adaptive runtime (`optiLib`): implements HTM in Go and provides required runtime mechanisms including retry and rollback (§ 5.4).

The source code patch choice, rather than a compiler transformation, is motivated by the desire to keep the developers in the loop. Using HTM without developers' knowledge can prove unwelcome because developers often demand full visibility into their programs. Developers are becoming performance and variance sensitive [56, 67, 83], and an accidental regression can become hard to diagnose. As a side effect, the choice of source-code patch demands us to be surgical — injecting large, complicated HTM-handling boilerplate code is a non-starter. Consequently, we perform `Lock()/Unlock()` operations replacements with API calls to HTM logic hidden in the `optiLib` open-source library and do so only in places where benefits of HTM are likely (e.g., we minimize the number of modified code locations using execution profiles).

4.1 GOCC Guarantees and Limitations

- GOCC will transform properly synchronized code (i.e., where every lock operation will have a corresponding unlock operation) into the equivalent code without changing the code's behavior. Code not meeting this criterion will be either not transformed, or transformed and its runtime behavior will be unchanged.
- GOCC considers only those lock-unlock pairs that seem to operate on the same lock within the same function — inter-procedural `Lock()/Unlock()` operations are disregarded. Note, however, that in a critical section protected by GOCC transformed lock can make arbitrary function calls. The requirement to have both `Lock()` and its matching `Unlock()` operation be present in the same procedure scope is only our implementation choice and pragmatic in nature. Over 70% of the locks we inspected met this criterion.
- GOCC makes no effort to identify critical sections or code reachability in the presence of reflection [10].
- GOCC, as implemented, does not *statically* detect HTM conflicts or capacity limitations (see § 5.2 for the details).

5 GOCC Design and Implementation

Before diving into the details of GOCC's design and implementation, we define some common terminology.

5.1 Terminology

Go's `sync` package provides two kinds of shared memory objects: `Mutex` and `RWMutex`. GOCC handles them both, but in the following sections, without the loss of generality, we will only use the term `Mutex` for simplicity. From an HTM transformation viewpoint, an `RWMutex` is no different from a `Mutex`, except `RWMutex` offers additional APIs for read-only accesses.

A *critical section CS* is all code regions protected by a pair of lock and unlock operations on the same mutex object `m` — the notation for calling lock/unlock operations on `m` is `m.Lock()/m.Unlock()` where `m` is referred to as a *receiver*. *Lock-point*, abbreviated with letter *L* (*Unlock-point* abbreviated with letter *U*), is a static location in the code where the `Lock()` (`Unlock()`) function is invoked on a `Mutex`. **LU-points** is a set of *L* and *U* points. **LU-pair** is a candidate pair of one *lock-point* paired with an *unlock-point*. In the runtime context, fastpath/HTM-path means the use of HTM, and slowpath/fallback-path means the use of the original lock.

We utilize the Abstract Syntax Tree (AST), program Control Flow Graph [85] (CFG), and Static Single Assignment (SSA) [35] form of program representation prevalent in the compiler literature. In a CFG, nodes are basic blocks [85] of straight-line code, and edges are control flow relationships among them. GOCC first transforms the source code to the AST form (which is also used for code transformation as described in § 5.3) and then to the SSA form for CFG construction.

5.2 Analyzer

The goal of the analyzer is to find as many LU-pairs as possible. The LU-pairs that protect HTM-incompatible critical sections (e.g., those including IO operations) must be pruned. This filtering serves two purposes: it reduces the number of code changes and non-beneficial HTM transformations. Complicated lock usage patterns, several Go language quirks, and pointer imprecision complicate the static analysis. A comprehensive call-graph analysis is vital because critical sections often contain function calls.

Conflicts: A sophisticated static analysis may detect whether transactions conflict. Answering this question, however, is unlikely to be valuable because developers typically do not use a lock if a conflict is impossible. Assuming conflicts happen, there is no easy way to statically determine whether transactions do not “typically” conflict. We do not try to solve this problem and leave conflict resolution to `optiLib`.

Capacity: Although one can perform static analysis to estimate the memory footprint of a critical section, it may not be possible if the bounds of a loop are unknown. Also, without knowing the target architecture's HTM capacity, it would be premature to filter out candidate critical sections this way. We leave the capacity-related decisions also to `optiLib`.

In the rest of this section, we, first, define the scope of our transformation (§ 5.2.2); then, describe the process of

<pre> 1 2 m := &sync.Mutex{} 3 m.Lock() 4 m.Unlock() </pre>	<pre> 1 l := OptiLock{} 2 m := &sync.Mutex{} 3 l.FastLock(m) 4 l.FastUnock(m) </pre>
---	--

Listing 1: Original lock-based code. **Listing 2:** Transformed HTM code.

matching a lock with an unlock operation within a code region assuming no lock nesting and no function calls in a CS; extend our analysis to include nested locks (§ 5.2.3); expand the analysis scope to CSs that may contain function calls (§ 5.2.4); detail special case of Go’s defer statement (§ 5.2.5); and finally discuss profile-based filtering (§ 5.2.6).

5.2.1 Scope of Transformation

To simplify the analysis, if a Lock() /Unlock() operation is executed in the middle of a basic block, we break such basic blocks in the CFG so that each lock-point begins a new basic block and each unlock-point ends a basic block. A single-entry single-exit (SESE) region [60] (simply *region*) of a CFG is our smallest granularity of lock transformation. A region is a subgraph of a CFG. Control reaching any basic block in a region is guaranteed to have already executed a designated entry basic block; control leaving from any basic block in the region is guaranteed to eventually pass through a designated exit basic block.

A function is the largest granularity of our lock transformation; a function always forms a region because all exits from a function are considered to go through a dummy basic block. This choice is pragmatic in nature since LU-pairs spanning multiple functions are uncommon.

Regions can be nested within one another. A Program Structure Tree (PST) organizes regions into a hierarchical tree [60]. We visit regions inside out from most-nested to least-nested. Appendix B in the extended version of this paper [94] describes the region identification and visiting strategies, which are not central to this paper.

5.2.2 Matching LU-pair in the Absence of Nested Locks

This subsection discusses analyzing a candidate region R .

LU-points in R may be operating on different locks, which should be pruned. Some lock (unlock) operations may escape R , without a corresponding unlock (lock) operation in R , which should also be pruned. Below, we formalize these aspects.

Definition 5.1 (Points-to set $\mathcal{M}(L)$ of a Lock point L). *Every lock-point (L) operates on some receiver mutex pointer p .³ Such a mutex pointer may point to one or more mutex objects allocated in the program. The set of all possible Mutex objects that p may point to in the program is the Points-to Set of L , denoted by $\mathcal{M}(L)$.*

³At the source level p can be either a pointer or an actual object value, but at the SSA level it is always a pointer.

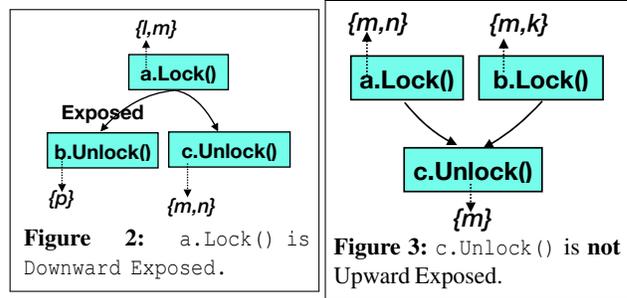


Figure 2: a.Lock() is Downward Exposed.

Figure 3: c.Unlock() is not Upward Exposed.

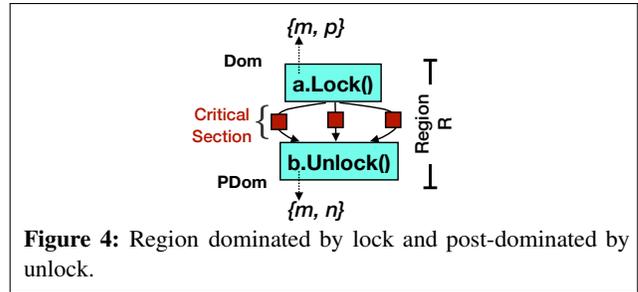


Figure 4: Region dominated by lock and post-dominated by unlock.

Similarly, the Points-to set of an Unlock point U is $\mathcal{M}(U)$. We employ Anderson’s flow-insensitive may-alias analysis [25] to obtain $\mathcal{M}(L)$ and $\mathcal{M}(U)$ on the whole program.

Definition 5.2 (Downward Exposed Lock-point (DELock)). *A lock-point, L , with points-to set $\mathcal{M}(L)$, is downward exposed in region R , if there exists at least one path from L to R ’s exit without any unlock-point on any mutex in $\mathcal{M}(L)$.*

Definition 5.3 (Upward Exposed Unlock-point (UEUnlock)). *An unlock-point, U , with points-to set $\mathcal{M}(U)$, is upward exposed in region R , if there exists at least one path from R ’s entry to U without a lock-point on any mutex in $\mathcal{M}(U)$.*

DELock identifies lock-points that *definitely* do not have any corresponding unlock-points in some execution paths in R ; and UEUnlock identifies unlock-points that *definitely* do not have corresponding lock-points in some execution paths in R .

Figure 2 exemplifies a downward exposed lock-point. Mutex pointer a ’s points-to set $\{l,m\}$, has an empty intersection with b ’s points-to set $\{p\}$; although it has a non-empty intersection with c ’s points-to set $\{m,n\}$. Figure 3 exemplifies an unlock-point that is *not* upward exposed. Mutex pointer c ’s points-to set $\{m\}$, has non-empty intersection with a ’s points-to set $\{l,m\}$ and b ’s points-to set $\{m,k\}$.

We eliminate all $DELock(R)$ and $UEUnlock(R)$ from the transformation in R . The remaining lock-points in R are the complement of $DELock(R)$, which is denoted by $\overline{DELock}(R)$. Similarly, the remaining unlock-points in R are the complement of $UEUnlock(R)$, which is denoted by $\overline{UEUnlock}(R)$.

Definition 5.4 (Feasible-HTM-Pair). *Let $L \in \overline{DELock}(R)$. Let $U \in \overline{UEUnlock}(R)$. L and U form a feasible HTM pair if all of the following conditions are true,*

- (1) $\mathcal{M}(L) \cap \mathcal{M}(U) \neq \emptyset$,
- (2) $(L \text{ DOM } U) \wedge (U \text{ PDOM } L)$,
- (3) The critical section $C \subseteq R$ guarded by L and U contains no LU-point X such that $\mathcal{M}(X) \cap (\mathcal{M}(L) \cup \mathcal{M}(U)) \neq \emptyset$, and
- (4) C contains no HTM-unfriendly instructions.

Condition (1) filters out those LU-points that are guaranteed to be operating on different `Mutex`s.

Condition (2) filters out infeasible control flows where unlock happens before lock and vice-versa. `DOM` and `PDOM` respectively represent dominator [85] and post-dominator [85] relationships in a CFG. Figure 4 shows an example, where all paths from lock-point `a.Lock()` are post-dominated by unlock-point `b.Unlock()`, whose all incoming paths are dominated by `a.Lock()`. Additionally, the set-intersection of the points-to set of mutex pointers $a = \{m, p\}$ and $b = \{m, n\}$ is non-empty. Any Feasible-HTM-Pair on L and U , forms an SESE-region by itself, where the entry basic block has L as its first instruction and the exit basic block has U as its last instruction. Condition (2) intuitively finds correct candidate LU-pairs in the absence of nested locks because if a lock operation L is performed on every path reaching any code in C and an unlock operation U is performed on every path exiting C , then LU must be operating on the same `Mutex`. Appendix A in the extended version of this paper [94] justifies our choice of `DOM/PDOM` relationships.

Condition (3) ensures that if we match an L with a U , there does not exist another lock-point or unlock-point in the same region that *may* operate on a `Mutex` in the same points-to set as that of L or U . The next subsection elaborates on lock nesting.

Condition (4) is an obvious requirement to ensure HTM does not abort. A region is unsafe if it contains any IO instructions.

Since we use “may alias” to match a lock-point with unlock-point, it is possible (but less likely) for our transformation to pair a lock with an unlock that may be operating on two different mutex objects at runtime. However, at runtime, we can obtain and memorize the address of the mutex object used at the lock-point, and compare it against the mutex object offered to the runtime at the unlock-point. In case of an address mismatch of the mutex objects used in the same LU-pair, we can abort the transaction and revert to a safe state and fall back to using the locks. A mismatch is impossible without nested locks because of the dominance and post-dominance relationship between the lock and unlock in an LU-pair.

5.2.3 Lock Nesting

Go supports nested locks, but reentrant [13] locks are not allowed. Condition (3) in Definition 5.4 allows nested locks but demands that they operate on disjoint `Mutex` objects.

HTM via Intel TSX allows nesting: if a nested transaction succeeds, hardware does not commit it until the outermost transaction commits. If a nested transaction fails, the control jumps to the starting code address of the nested transaction.

<pre> 1 a.Lock() //outer region start 2 3 4 b.Lock() // inner region start 5 b.Unlock() // inner region end 6 7 a.Unlock() //outer region end </pre>	<pre> 1 a.Lock() 2 3 l := OptiLock{} 4 l.FastLock(b) 5 l.FastUnlock(b) 6 7 a.Unlock() </pre>
--	--

Listing 3: Nested Locks.

Listing 4: HTMized.

<pre> 1 a.Lock() //outer region start 2 3 4 b.Lock() // inner region start 5 a.Unlock() // inner region end 6 7 b.Unlock() //outer region end </pre>	<pre> 1 a.Lock() 2 3 l := OptiLock{} 4 l.FastLock(b) 5 l.FastUnlock(a) 6 7 b.Unlock() </pre>
--	--

Listing 5: Hand-over-hand lock.

Listing 6: HTMized.

This facility allows us to safely transform locks into HTM even when they are nested.

Condition (3) in Definition 5.4 disqualifies a candidate LU-pair from the transformation in region R if there exists any other lock or unlock point whose lock/unlock operation *may* be operating on the same mutex as those in the LU-pair.

As an example, in Listing 3, assume the mutex pointers a and b point to the same points-to set. When inspecting the “inner region”, we find only one LU-pair, which obeys all Feasible-HTM-Pair conditions in Definition 5.4. Consequently, the lock usage on b in the inner region can be transformed to HTM. When inspecting the “outer region”, however, we see conflicting LU-points, and hence the locking operations on a will not be transformed. The resulting transformed code is shown in Listing 4, which is correct.

This approach complicates hand-over-hand locking [33, 62], sometimes used in the concurrent linked-list traversal, shown in Listing 5. As before, assume all four LU-points have a non-empty intersection of their points-to sets. When inspecting the inner region, the LU-pair `b.Lock()` and `a.Unlock()` passes all tests in Definition 5.4. Hence, they will be, incorrectly, paired and transformed to use HTM, as shown in Listing 6. This transformation violates the programmer’s intention. Subsequently, when visiting the outer region, condition (3) is violated, and hence the outer LU-pair will not be transformed. One could have discarded the transformation of the inner region when the conflict is visible in the enclosing region. However, we cannot distinguish this incorrect pairing from the correct pairing in the previous case. Our solution is to always apply the transformations on the candidates found in inner regions, and handle mismatches at runtime via HTM aborts iff executing on the fastpath. As mentioned at the end of § 5.2.2, a mismatch is easy to recognize at runtime by, first, making `FastLock()` store the address of the `Mutex` used at the lock-point in a field in `OptiLock` and, second, checking whether the `Mutex` passed to `FastUnlock()` is the one present in `OptiLock`. The transactional abort is needed (and possible) only on the fastpath. Appendix C in the extended version of this paper [94] details the correctness of transforming nested locks into HTM via GOCC.

5.2.4 Critical Sections with Function Calls

When the critical section protected by a candidate LU-pair contains function calls, we need to extend the analysis beyond the current function. Conditions (1) and (2) in Definition 5.4 are local to R . Conditions (3) and (4) require inter-procedural analysis.

We need to ensure that the transitive-closure of all code regions protected by a candidate LU-pair, including the blocks reachable via function calls, neither contains any HTM-unfriendly instructions nor contains any LU-points whose points-to set may overlap with the points-to sets of L or U . Otherwise, we discard the candidate LU-pair.

To accomplish this, we first build a static call graph using rapid type analysis [7, 26]. Next, we precompute summary information for each function on its own without its transitive closure of function calls; the summary contains (a) the fit of the function for HTM based-execution (i.e., no HTM-unfriendly instructions), and (b) the union of all points-to sets of all LU-points in the function, denoted by \mathcal{P} .

For a candidate LU-pair meeting all the conditions in Definition 5.4 within the region R , we proceed to do inter-procedural analysis. Let F^* be the transitive closure of all the function calls invoked inside the critical section $C \subseteq R$ protected by a candidate LU-pair. LU-pair is discarded if (a) $\exists F \in F^* . s.t. F$'s summary contains HTM-unfriendly instructions or (b) $\exists F \in F^* . s.t. \mathcal{P} \cup (\mathcal{M}(L) \cup \mathcal{M}(U)) \neq \emptyset$. The former is simply the condition (4) expanded to all functions, and the latter is condition (4) expanded to all functions. We note that nested locks discussed in § 5.2.3 can be in different functions.

5.2.5 The defer Statement

The `defer` [49] statement in Go, introduced in § 2, needs special attention. Go defers the execution of functions prefixed with the `defer` keyword to the calling function's return point. The presence of `defer Unlock()` complicates our CFG-based dominance/post-dominance analysis. Deferred unlocks extend the critical sections till function exit points. Listing 7 shows a legal Go code, where the `defer m.Unlock()` appears even before the call to `m.Lock()`. Condition (2) in Feasible-HTM-Pair will treat this pair as an invalid candidate for transformation because the lock-point does not dominate the unlock-point.

We address this case by interpreting `defer m.Unlock()` in a CFG by (a) introducing a synthetic `m.Unlock()` statement at the end of each basic block that returns control from the function, and (b) discarding the presence of `m.Unlock()` in its original position during the analysis. This allows us to reuse the previously described dominance relationship. During transformation, however, GOCC simply replaces a `defer Unlock()` with a `defer FastUnlock()` in its original place, as shown in Listing 8.

Multiple `defer` calls are executed in a last-in first-out (LIFO) order of encountering the `defer` statement at runtime.

```
1 func DeferExample() {           1 func DeferExample() {
2                               2     l := OptiLock{}
3     m := &sync.Mutex{}         3     m := &sync.Mutex{}
4     defer m.Unlock()           4     defer l.FastUnlock(m)
5     m.Lock()                   5     l.FastLock(m)
6     // critical section        6     // inside HTM
7 }                               7 }
```

Listing 7: defer Unlock.

Listing 8: HTMized.

This complicates the dominance and post dominance relationship; for simplicity, we discard any function that contains multiple `defer Unlock()` statements. We found none in the packages used in our evaluation.

5.2.6 Profiles to Filter Hot Critical Sections

Profiling is a built-in feature in Go, which takes callstack samples via timer [48] or hardware performance counter [30] interrupts. One can take CPU profiles of a go program either at launch time by simply passing a `-cpuprofile` flag or from an already running program, for a specified duration, by accessing an exposed profiling port.

Go profiles are in the `pprof` format.

The `pprof` Go package [48] allows us to programmatically navigate the callstack samples presented as weighted call graphs, where the nodes represent functions and edges represent caller-callee relationships. The functions are annotated with their inclusive and exclusive execution times.

When profiles are available, we use them to filter the regions where negligible execution time is spent, even before applying the static analysis. In fact, this is the first filtering step we perform before making the aforementioned LU-pair identification. We treat any critical section (including the entry and exit) where the aggregated execution time is less than 1% of the total execution time as insignificant.

5.3 Transformer

Since our end product is a code patch, we perform the transformation on the AST form of the program. Go AST can be serialized into source code via `Go format` [8] package. To this end, the transformer maps the candidate set of LU-pair operations found during the SSA-based analysis phase (described in § 5.2) to AST nodes [6]. It then replaces the LU-pair operations with calls to `FastLock()/FastUnlock()` in `optiLib`. It also passes the original `Mutex` object as a pointer to the calls to `FastLock()/FastUnlock()` since the underlying lock object is necessary for lock elision (fastpath) as well as for slowpath. Figure 5 shows an example AST transformation. The transformation itself is mechanical but challenging. In the following sections, we discuss several Go features that pose special challenges in transforming the AST.

Go pointer vs. value: Go syntax does not distinguish accessing a field of a composite type (e.g., `struct`) via an object-pointer or an object-value; both use the same

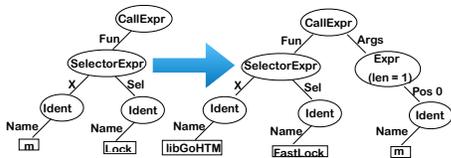


Figure 5: Example of AST transformation from `m.Lock()` to `optiLib.FastLock(m)`. Some AST nodes are omitted for brevity.

```

1 // pointer form
2 m := &sync.Mutex{}
3 m.Lock()
4 m.Unlock()
5
6 // value form
7 n := sync.Mutex{}
8 n.Lock()
9 n.Unlock()

```

Listing 9: Both Mutex pointer and Mutex value `n` invoke `Lock/Unlock` using the same `FastLock()/FastUnlock()` but dot dereferencing operator. `&n` to `FastLock()/FastUnlock()`.

AST dot operator as exemplified in Listing 9. However, `FastLock()` and `FastUnlock()` must receive a pointer to the Mutex object to perform the elision correctly. Hence, if the LU-pair uses a Mutex value, its address needs to be passed to `FastLock()/FastUnlock()`, and if the LU-pair uses a Mutex pointer, it should be passed as is.

We address this issue in the transformer by querying the type information [9] for each receiver object subject to transformation. If the receiver identifier is a Mutex value type, we insert the additional address-of operator before the Mutex identifier in the AST and pass it to `FastLock()/FastUnlock()`. If the receiver identifier is a pointer to a Mutex type, we pass it as is.

Go anonymous fields: Go allows programmers to define a struct that has fields without names. For instance, Listing 11 shows a struct `AStruct` that has an anonymous `*sync.Mutex` field. Operations on this anonymous mutex are performed by simply using the name of the enclosing struct variable as shown on Line 8. Hence, our transformation needs to be cognizant about whether the original LU-pair operations are performed on an anonymous mutex.

By inspecting the type information [9] of the *access path* [66] used to invoke the lock/unlock operation in the AST, we determine whether or not the operation is performed on an anonymous mutex field. Upon determining the operations to be on an anonymous mutex, we pass the address of the anonymous Mutex to `optiLib` by simply suffixing the operation access path with `Mutex` as shown in Listing 12, Line 8 (where access path simply consists of variable `a`). This transformation composes with the previously described pointer vs. value operations.

Anonymous goroutines: Go supports anonymous goroutines [2], which are nested inside other functions as

```

1 type AStruct struct {
2   x int //not anonymous
3   *sync.Mutex //anonymous
4 }
5 func main() {
6   a := AStruct{}
7   a.Lock()
8   a.Unlock()
9 }

```

Listing 11: Locking on an unnamed field of a struct.

```

1 m := &sync.Mutex{}
2 for i:=0;i<10;i++ {
3   go func() {
4     l := OptiLock{}
5     m.Lock()
6     // CS
7     m.Unlock()
8   }()
9 }
10 //wait all

```

Listing 13: Anonymous goroutines create concurrent units needed for the transformation of execution on anonymous should be placed in the innermost function scope.

Listing 14: The `OptiLock` outlines create concurrent units needed for the transformation of execution on anonymous should be placed in the innermost function scope.

shown in Listing 13; these goroutines run concurrently. Our transformation introduces a new `OptiLock` variable in transformed functions. `OptiLock`'s declaration should be in the scope that encloses both `Lock` and `Unlock` operations, but it should not be shared by other concurrent executions because it maintains goroutine-specific state. Hence, we make `OptiLock` a variable in the stack of each goroutine. We add the declaration to the innermost function body as shown on line 4 in Listing 14. A bottom-up AST walk from LU-pair to be transformed allows us to easily detect the innermost enclosing anonymous function scope.

5.4 Adaptive HTM Runtime: `optiLib`

`optiLib` implements all the intelligent runtime control needed to perform HTM in lieu of locks. It is in charge of starting and committing transactions in critical sections, as well as falling back to the lock when necessary. It is responsible for inter-operating with locking operations on the same mutex that may not be transformed to use HTM. In the event of aborts, it is responsible for determining the cause of the abort and deciding whether and how many times to retry. If, accidentally, the code rewriting matches lock-point with a programmer-unintended unlock-point, `optiLib` is responsible for detecting and recovering from the mistake. Finally, it is responsible for understanding and dynamically adjusting to changing contention.

We implement `optiLib` using TSX [5] for Intel platforms. `optiLib` is carefully implemented to ensure correctness under all circumstances. Equally important, it is implemented with the utmost attention to performance. Every instruction and its placement are carefully planned to minimize any overhead of its own. `optiLib` uses Intel RTM; it does not use the Hardware Lock Elision (HLE) [1] because it does not provide

the fine-grained control we need.

`optiLib` introduces a data structure: `OptiLock`, which has two fields: a boolean `slowPath` and a `*sync.Mutex lkMutex`. `slowPath` is set to `true` if the lock is not elided at runtime. `lkMutex` always holds the address of the fine-grained lock being elided. `OptiLock` supports `FastLock()/FastUnlock()` operations, both need a `*sync.Mutex` argument, which is the mutex receiver object being elided at the original call sites of `Lock()/Unlock()`. Try locks and timed locks [31, 78] are absent in Go; it is trivial to support them in `optiLib`.

The `FastLock()` implementation uses sophisticated mechanisms described previously by others [23, 32, 64] to interoperate slow and fast paths concurrently. Stated succinctly, the `FastLock()`, waits for the fine-grained lock to be available before starting the hardware transaction; on starting a transaction, it first checks if the fine-grained lock is already held and unconditionally aborts if it is already held; if it is not held, the act of checking adds the lock word to the transaction read-set, and hence, if a concurrent execution on the slowpath acquires the same lock during the transaction, the fastpath immediately aborts ensuring mutual exclusion. Any two threads in the fastpath can run concurrently as long as they do not conflict in their memory accesses.

Reading the internals of the original `sync.Mutex` object is straightforward and has no performance penalty; `FastLock()` simply de-references the first word of the `Mutex` pointer passed into the function, which contains the lock status.

The `FastUnlock()`, based on `slowPath` value, either commits the transaction or invokes the unlock on the mutex object. It also safeguards against accidental incorrect code patches by ensuring that the mutex object passed into `FastLock()` and stored in the `lkMutex` field of `OptiLock` matches the mutex object presented to `FastUnlock()`. In case of a mismatch, `FastUnlock()` restores safety by aborting the transaction (iff on fastpath), and subsequently enforces the slowpath.

5.4.1 Dynamic adjustment via perceptron

It would not be fruitful to attempt HTM if doing so has already proved to be unsuccessful for whatever reason. `GOCC` learns and adapts to HTM behavior and decides whether to use HTM for the already transformed LU-points, the fallback being the original lock. For this purpose, `GOCC` uses a featherlight, hardware-inspired “hashed perceptron” [84].

The hashed perceptron predictor hashes feature weights into one or more tables. Then at the prediction time, it uses indexes to access feature weights from the tables and adds up all the relevant weights. If the sum exceeds a threshold, the prediction will be regarded as positive (e.g., “HTM should be taken”). Otherwise, the result will be viewed as negative (e.g., “HTM should not be used”). The weights will be updated based on the correctness of the predictions.

If operations on a given `Mutex` have been HTM-friendly/unfriendly, we want to utilize this information.

Similarly, if a code location has been HTM-friendly/unfriendly irrespective of the `Mutex` used, we want to use this information as well. Hence, the two input features for the perceptron are the `Mutex` and the calling context [24, 50] of lock/unlock invocation. The address of the `Mutex` serves as the `Mutex` feature, and the address of `OptiLock` serves as a unique identifier for the calling context feature. Updating the same perceptron weight for the `Mutex` feature by different goroutines would create a conflict (and potentially a performance collapse). Hence, we instead XOR the `Mutex` address with the address of the `OptiLock` to produce a conflict-free feature input.

Our perceptron implementation creates two 4K-entry arrays as the global weight tables (GWT). The weights take an integer number ranging from -16 to 15. At runtime, `FastLock()` and `FastUnlock()` functions index into GWT by taking the lower-12 bits of the two features. Perceptron operations are done outside the transaction. The updates and reads from GWT are lock-free but racy — perfection is not required here, but high-performance is necessary. Experiment results from § 6.2 show the effectiveness of perceptron learning in protecting against poor HTM performance.

Perceptron weight update: Perceptron weight updates happen in the `FastUnlock()` function after successfully finishing the critical section, whether on fast and slow path.

If the perceptron decides that the lock should be used, there will be no update to the weights as the lock will always succeed. When the perceptron indicates to use the HTM and the execution finishes on the fastpath, the corresponding weight in the cell will be increased (because the perceptron makes a correct decision, it should be encouraged to use the HTM more frequently). On the other hand, if perceptron determines to use HTM, but HTM fails and falls back to slowpath, the weights will be decreased (because HTM does not work for the current call, perceptrons should be penalized for incorrect recommendation to improve future predictions).

Weight decay: We keep a counter, in each cell in GWT, to record the number of lock calls that go to the slowpath directly as a result of perceptron decision. If a lock has been used consecutively for a certain number of times and exceeds the threshold, we reset the weight of the perceptron cell and subsequently try HTM. Without this reset, perceptron would get stuck on the slowpath, preventing the benefits of the HTM execution in the future. We set this threshold to 1000 continuous decisions. Appendix D in the extended version of this paper [94] summarizes our `FastLock()/FastUnlock()` implementations including the perceptron logic.

5.4.2 Alleviating HTM overhead

HTM brings overhead for very short critical sections as described in § 2 above, even under single-core execution. `optiLib` avoids using HTM if it recognizes a single OS-thread in a Go process. `optiLib` employs `runtime.GOMAXPROCS(0)` API for this purpose.

repo	stars	contrib utors	com mits	LoC	Lock points	Unlock points	violates dominance	Candi date pairs	unfit for HTM	Nested alias locks	Transformed Pairs w/o profiles	Transformed Pairs w/ profiles
						total (defer)			intra/interproc		intra/interproc	total (defer)
tally	450*	27	95	2.4k	54	56 (28)	2	52	2/29	0/0	21 (14)	7 (7)
zap	4.5k*	7	163	3.3k	8	8 (4)	0	8	3/0	0/0	5 (1)	6 (0)
go-cache	11.6k*	71	322	18k	96	230 (6)	68	28	0/2	0/0	26 (4)	10 (2)
fastcache	59k*	40	673	33k	24	24 (2)	2	22	2/2	0/0	18 (0)	7 (4)
set	967*	8	48	2.4k	16	16 (10)	0	16	0/2	0/0	14 (8)	8 (2)

Table 1: Go package characteristics and their behavior using GOCC

6 GOCC Evaluation

We evaluate GOCC on an 8-core ($\times 2$ -way SMT [87]) Intel Coffee Lake CPU with a total 32GB memory, running Linux 5.4.0. The CPU has 32KB L1I and L1D cache, 256KB L2 cache, and 16MB L3 cache. The Go version is 1.15.2.

Table 1 shows the list of applications and libraries we employ. In the absence of standard benchmarking for Go, we selected packages that are popular open-source Go projects (column 2 in Table 1), focus on high performance, utilize lock-based Go concurrency, and provide thread-safe APIs. In particular, Zap and Tally are foundational logging and metrics collection packages used in production go programs by many organizations. Additionally, since we evaluate the projects using their own benchmark suites (more on this below), we only selected projects that feature concurrent benchmarks or whose benchmarks could be straightforwardly converted to be concurrent.

From a static analysis viewpoint, we see that all applications contain several locks. Defer unlocks are common (column 7). The “violates dominance” column shows how many LU-points were discarded since they did not meet the dominance relationship. The number is typically low except for `go-cache`, which has several functions with the repeating pattern of unlocks that do not post-dominate the candidate lock. The “candidate pairs” column shows how many LU-points remain for further analysis. Each column to the right progressively shows the reasons for which a candidate LU-pair was rejected. Rejection due to nested aliased locks is not found in these packages. The second-to-last column shows how many LU-pairs were finally rewritten to use `OptiLock`, including how many of them contain `defer Unlock()`. The last column shows the numbers after we retain only those locks where the functions contain at least 1% of execution time in execution profiles. Overall, GOCC transforms several LU-pairs in each application. Using profiles significantly reduces the number of transformed LU-pairs.

We run all the benchmarks within each repository five times and report the median. We believe the benchmarks accompanying the code best represent its desired characteristics. As some benchmarks are written for a single thread setting, we rewrite them to introduce concurrency to utilize HTM-enabled parallelism fully. We adopt the standard testing package from Go [16], which runs each benchmark for a certain amount of time and reports the throughput as nanoseconds per operation. We wrap the benchmark codes with `RunParallel` [4] helper

function to get parallel performance if it was not already done so. Using more CPU cores, ideally, increases throughput (i.e., reduces average nanoseconds per operation). Then we compare the throughput from locks vs. HTM — a positive percentage means GOCC’s rewrite did better, and a negative percentage means the baseline did better. We vary the number of CPU cores available for benchmarks from 1 to 8. Unlike HPC codes that run on all cores on a server, Go services often use 2-4 cores.

6.1 Results on Popular Go Programs

We categorize the benchmarks in each package into two groups:

1. **Concurrency non-sensitive** benchmarks either have no locks or do not spend much time in critical sections, or our transformation does not result in any performance difference. For these benchmarks, we only show the aggregate (geomean) results unless noted otherwise. They appear as “non sensitive” in our charts, and the number in the parenthesis indicates how many benchmarks are in this group.
2. **Concurrency sensitive** benchmarks exercise modified locks non-trivially. We might have impacted them positively or negatively. For these, we present data from each benchmark and also present an aggregate result (“sensitive” in our charts).

The “all” part of our charts is the geomean taken over all benchmarks. Sometimes this number looks small because of a large number of non-sensitive benchmarks.

In what follows, we provide details of performance evaluation on the aforementioned Go packages. The total number of benchmarks is large; hence, we dive deep only into benchmarks with surprisingly good speedups.

Tally [88] is a fast, buffered stats collection library and Figure 6 shows its results. For the `HistogramExists` benchmark, GOCC achieves more than 660% speedup on 8 cores reducing the original time per operation from 65 ns/ops down to around 8.47 ns/ops at 8 cores. Moreover, the HTM delivers scalable performance. This benchmark uses a `Mutex` lock on a read-only `Exists` operation, and hence, is a natural candidate to demonstrate speedup as HTM eliminates the unnecessary serialization. Conversely, the baseline has a scalability collapse, where the time per operation increases from 20.4 ns/ops to 65 ns/ops for 1 to 8 cores. `ScopeReporting1` holds three independent `RWMutexes` at different points in time and accesses read-

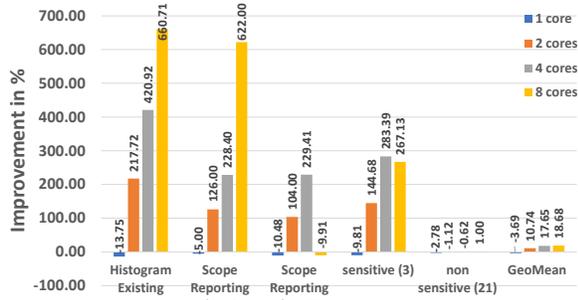


Figure 6: Results on Tally with different core numbers.

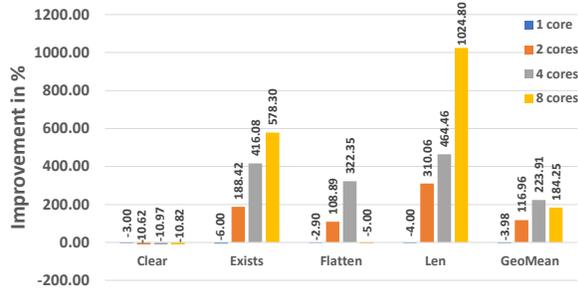


Figure 8: Results on concurrent set with different core numbers.

only data. However, since the `RWMutex` also involves a counter increment and a decrement, its overhead as a result of cache invalidation does not scale well. Thus, even eliding `RWMutex` proves highly beneficial. The speedup for `ScopeReporting10` is lower than that for `ScopeReporting1` because it performs 10x more work inside the critical section. Overall, in the sensitive group, we see a 10% performance drop with a single CPU but 145%, 283%, and 267% improvements with 2, 4, and 8 CPUs, respectively. In the non-sensitive group, the overall performance drop is within the margin of error. Among all the 27 benchmarks of tally, we see up to 18.7% speedup at 8-CPU.

`go-cache` [12] is an in-memory key-value store. It contains benchmarks that exercise repeatedly accessing the same item in a small map. The benchmarks contain both non-cached accesses, similar to how go programmers often use a map, and cached accesses provided by the `go-cache` layer to demonstrate the effectiveness of the library. All benchmarks employ `RWMutex`s for concurrent map read access. Unlike the rest of the use cases, the benchmark files themselves contain locks, which GOCC transforms into using HTM.

Figure 7 shows our empirical results. GOCC speeds up four benchmarks in `go-cache` that were directly accessing the map without the library-provided cache. In each case, we can see more than 100% speedup; the biggest speedup is 742%. The speedups come from eliminating contended atomic operations involved in entering and exiting from a reader lock. The performance scales well with increased parallelism because while the lock-based approach incurs more and more contention, the HTM approach remains conflict-free throughout. The other benchmarks, the majority of which employ the `go-cache`,

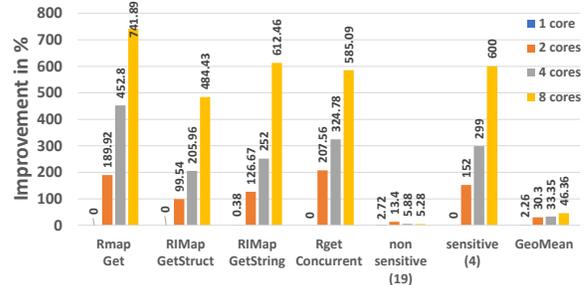


Figure 7: Results on go-cache with different core numbers. (benchmark names reflect abbreviated names of go-cache’s benchmark functions).

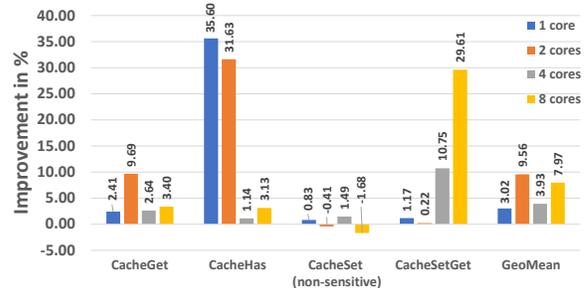


Figure 9: Results on fastcache with different core numbers.

are mildly improved, but more importantly, they were not degraded as a result of transformation via GOCC.

`go-datastructures` [21] is a collection of performant, thread-safe data structures. We apply GOCC on the `set` subdirectory, which contains concurrency benchmarks. The results are shown in Figure 8. The `Len` benchmark computes the length of the set, and it is sped up by $\sim 1000\%$ in the 8 cores setting. `Len` has a short critical section that has a higher entry and exit cost due to atomic operations when using `arWMutex`. HTM performance shows scalability since the HTM version remains conflict-free, whereas the lock-based version collapses with increased contention. The `Exists` benchmark is similar to `Len`, where each goroutine searches one item in a set containing only one item. It scales almost as well as `Len`, but more work is done in the critical section, which amortizes `RWMutex`’s overhead, and slightly reduces HTM’s advantage. The `Flatten` benchmark reads 50 elements from a shared map into a private array, with a layer of caching that eliminates repeated map scanning. It holds a `Mutex` to serialize concurrent accesses to the map/cache. The HTM version avoids the serialization and shows scalable performance for 1-4 cores. At 8 cores, the number of conflicts resulting from updating the cache rises, which makes perceptron not use the HTM, and hence there is no speedup. The `Clear` benchmark has true conflicts, and there is no speedup, but the HTM does not significantly degrade the performance. Overall, utilizing GOCC results in more than 100% geomean performance gain while introducing less than 4% slowdown in a single core setting.

`Zap` [17] is a library that implements fast and structured logging in Go. Being a logging library, it has several IO operations,

and hence GOCC rewrote fewer locks. Compared with other repositories, the improvement on zap is relatively mild. Due to arguably mild speedups on Zap, a large number of benchmarks, we omit a deeper analysis of Zap results. Slowdowns are rare, the biggest being 7%. Overall, we observed a mild $\sim 4\%$ geometric mean speedup with the best case 28% speedup.

Fastcache [18] is a fast, scalable, in-memory cache. The transformed code delivers a maximum of 35.60% speedup and a geomean of 15.65% speedup across all benchmarks.

In the **CacheGet** benchmark, goroutines repeatedly invoke the `Get` function, which uses an `RWMutex` to protect a shared map. `Get` has inter-procedural nested but non-conflicting locks, all of which are transformed into HTM. `Get` looks up a key in the map and returns a value blob. The critical section of `Get` contains a few atomic add instructions, which update shared variables. Transactional conflicts on the shared atomic adds are fewer at low core numbers, and the speedup is visible; however, at larger core counts, the conflicts increase, and the speedup vanishes. Fortunately, the perceptron kicks in and avoids any performance collapse.

The **CacheHas** benchmark is virtually the same as **CacheGet**, but its critical section is shorter since it does not return a populated value buffer. Hence, the speedups are higher due to fewer conflicts, but it follows the same performance pattern as **CacheGet**.

In the **CacheSetGet** benchmark, each goroutine has two loops: the first loop repeatedly invokes `Set` and the second loop repeatedly invokes `Get`. The `Set` function, which inserts a key-value pair into the map, may raise a `panic` if certain constraints are violated. Hence, GOCC does not modify a `Lock()` present in `Set`. The `Get` function is already described previously.

Since all goroutines first attempt `Set`, where Go's default locks are being used, the runtime recognizes it as a starved mutex and takes away the time slice of some of the goroutines. This runtime behavior reduces the number of lock contenders and, as a result, a few goroutines monopolize the lock. These goroutines quickly finish their series of `Set` operations and proceed into calling `Get` in a loop. The contention is lower on `Get` also since the load is now split between `Get` and `Set` with some goroutines on hold. The net effect is a high throughput for the whole benchmark.

It is worth noting that the only other benchmark which invokes the `Set` function is the non-sensitive benchmark **CacheSet**. Even though **CacheSet** exhibits no performance improvement, and **CacheGet** shows mild performance improvement, their composition in **CacheSetGet** leads to secondary effects causing much higher performance gain at higher core counts.

6.2 Perceptron Evaluation

We assess the effectiveness of perceptron using the **Tally** benchmarks. We compare the performance with and without the perceptron machinery. In the absence of the perceptron,

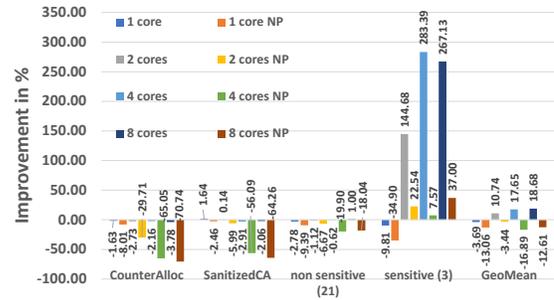


Figure 10: Results on **Tally** to show the effectiveness of perceptron. NP \Rightarrow No Perceptron.

we always attempt HTM. In the results presented in Figure 10, we can observe that the perceptron is effective in eliminating any performance loss. For example, **CounterAllocation** and **SanitizedCounterAllocation** are HTM-unfriendly benchmarks and cause aborts frequently. Perceptron quickly learns to move away from HTM and keeps using the slowpath. Therefore, there is minimal performance loss for the perceptron case.

Finally, we setup a synthetic benchmark — a conflict-free critical section with 1000 counter updates — to evaluate the overhead of the perceptron machinery. We measured the perceptron prediction overhead to be 0.65% and weight update overhead to be 0.73% for a total of only 1.38%.

7 Conclusions

GOCC is a source-to-source transformation tool to speed up lock-based pessimistic concurrency control in Go programs with Hardware Transactional Memory. GOCC combines thorough static analysis with intelligent runtime control to expose additional parallelism available in Go programs. GOCC keeps the developer in the loop, minimizes code changes via execution profiles, and targets only those critical sections that are likely to improve with HTM. The experimental results from real-world Go packages show that GOCC delivers significant (up to $10\times$), scalable performance for concurrent Go code that uses locks while exhibiting rare and relatively small slowdowns.

8 Availability

GOCC is available as an open-source tool [3].

9 Acknowledgement

This material is based upon work supported in part by the National Science Foundation under Gran No. 1763699, 1717779, 1563935. We thank our shepherd Michael Spear and the anonymous reviewers for their feedback.

References

- [1] GitHub - linux4life798/safetyfast: An Go library of synchronization primitives to help make use of hardware transactional memory (HTM). <https://github.com/linux4life798/safetyfast>.
- [2] Go by Example: Closures. <https://gobyexample.com/closures>.
- [3] Go Optimistic Concurrency Control (GOCC). <https://github.com/uber-research/GOCC>.
- [4] Golang benchmark RunParallel API. <https://golang.org/pkg/testing/#B.RunParallel>.
- [5] Intel 64 and IA-32 Architectures Optimization Reference Manual. <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf>.
- [6] Package astutil. <https://godoc.org/golang.org/x/tools/go/ast/astutil>.
- [7] Package callgraph. <https://pkg.go.dev/golang.org/x/tools/go/callgraph>.
- [8] Package format. <https://golang.org/pkg/go/format/>.
- [9] Package go/types. <https://golang.org/pkg/go/types/>.
- [10] Package reflect. <https://golang.org/pkg/reflect/>.
- [11] Package sync. <https://golang.org/pkg/sync/>.
- [12] patrickmn/go-cache: An in-memory key:value store/cache (similar to Memcached) library for Go, suitable for single-machine applications. <https://github.com/patrickmn/go-cache>.
- [13] Reentrant Mutex. https://en.wikipedia.org/wiki/Reentrant_mutex.
- [14] Solved: A low background number of - Intel Community. <https://community.intel.com/t5/Software-Tuning-Performance/TSX-conflict-aborts-for-single-threaded-applications/m-p/983986#M3190>.
- [15] sync: Mutex performance collapses with high concurrency. <https://github.com/golang/go/issues/33747>.
- [16] testing - The Go Programming Language. <https://golang.org/pkg/testing/>.
- [17] uber-go/zap: Blazing fast, structured, leveled logging in Go. <https://github.com/uber-go/zap>.
- [18] VictoriaMetrics/fastcache: Fast thread-safe inmemory cache for big number of entries in Go. Minimizes GC overhead. <https://github.com/VictoriaMetrics/fastcache>.
- [19] Why Locking in Go much slower than Java? <https://stackoverflow.com/questions/39815723/why-locking-in-go-much-slower-than-java-lots-of-time-spent-in-mutex-lock-mut>.
- [20] Why Use the Go Language for Your Project? <https://nix-united.com/blog/why-use-the-go-language-for-your-project/>.
- [21] Workiva/go-datastructures: A collection of useful, performant, and threadsafe Go datastructures. <https://github.com/Workiva/go-datastructures>.
- [22] Yehuda Afek, Amir Levy, and Adam Morrison. Software-Improved Hardware Lock Elision. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, page 212–221, New York, NY, USA, 2014. Association for Computing Machinery.
- [23] Yehuda Afek, Alexander Matveev, Oscar R. Moll, and Nir Shavit. Amalgamated lock-elision. In Yoram Moses, editor, *Distributed Computing*, pages 309–324, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
- [24] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, page 85–96, New York, NY, USA, 1997. Association for Computing Machinery.
- [25] Lars Ole Andersen. Program Analysis and Specialization for the C Programming Language. Technical report, University of Copenhagen, 1994.
- [26] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, page 324–341, New York, NY, USA, 1996. Association for Computing Machinery.
- [27] Vincent Blanchon. Go: Mutex and Starvation. <https://medium.com/a-journey-with-go/go-mutex-and-starvation-3f4f4e75ad50>, Sep 2019.
- [28] Vincent Blanchon. Go: How to Reduce Lock Contention with the Atomic Package. <https://medium.com/a-journey-with-go/>

[how-to-reduce-lock-contention-with-the-atomic-package-ba3b2664b549](#), Aug 2020.

- [29] Irina Calciu, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. In *Transact 2014 Workshop. ACM*, page 54, 2014.
- [30] Milind Chabbi. pprof++: A Go Profiler with Hardware Performance Monitoring. <https://eng.uber.com/pprof-go-profiler/>, May 2020.
- [31] Milind Chabbi, Abdelhalim Amer, Shasha Wen, and Xu Liu. An Efficient Abortable-locking Protocol for Multi-level NUMA Systems. In Vivek Sarkar and Lawrence Rauchwerger, editors, *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*, pages 61–74. ACM, 2017.
- [32] Milind Chabbi and John Mellor-Crummey. Contention-Conscious, Locality-Preserving Locks. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '16, New York, NY, USA, 2016*. Association for Computing Machinery.
- [33] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-Volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '14*, page 433–452, New York, NY, USA, 2014. Association for Computing Machinery.
- [34] Keith Chapman, Antony L. Hosking, and J. Eliot B. Moss. Hybrid STM/HTM for Nested Transactions on OpenJDK. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016*, page 660–676, New York, NY, USA, 2016. Association for Computing Machinery.
- [35] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.
- [36] Luke Dalessandro, Francois Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVI*, page 39–52, New York, NY, USA, 2011. Association for Computing Machinery.
- [37] Dave Dice, Maurice Herlihy, Doug Lea, Yossi Lev, Victor Luchangco, Wayne Mesard, Mark Moir, Kevin Moore, and Dan Nussbaum. Applications of the adaptive transactional memory test platform. In *3rd ACM SIGPLAN Workshop on Transactional Computing*, pages 1–10, 2008.
- [38] Dave Dice, Alex Kogan, and Yossi Lev. Refined Transactional Lock Elision. *SIGPLAN Not.*, 51(8), February 2016.
- [39] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. Adaptive Integration of Hardware and Software Lock Elision Techniques. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14*, page 188–197, New York, NY, USA, 2014. Association for Computing Machinery.
- [40] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. Adaptive Integration of Hardware and Software Lock Elision Techniques. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14*, page 188–197, New York, NY, USA, 2014. Association for Computing Machinery.
- [41] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. *SIGARCH Comput. Archit. News*, 37(1):157–168, March 2009.
- [42] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pages 157–168, 2009.
- [43] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the 20th International Conference on Distributed Computing, DISC'06*, page 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.
- [44] Nuno Diegues and Paolo Romano. Self-Tuning Intel Transactional Synchronization Extensions. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 209–219, Philadelphia, PA, June 2014. USENIX Association.
- [45] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and Limitations of Commodity Hardware Transactional Memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, PACT '14*, page 3–14, New York, NY, USA, 2014. Association for Computing Machinery.
- [46] Google. Effective Go - The Go Programming Language. https://golang.org/doc/effective_go.html.
- [47] Google. Kubernetes. <https://kubernetes.io/>.

- [48] Google. Profiling Go Programs. <https://blog.golang.org/pprof>.
- [49] Google. The Go Programming Language Specification. https://golang.org/ref/spec#Defer_statements.
- [50] Susan L Graham, Peter B Kessler, and Marshall K McKusick. An execution profiler for modular programs. *Software: Practice and Experience*, 13(8):671–685, 1983.
- [51] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: A Benchmark for Software Transactional Memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, page 315–324, New York, NY, USA, 2007. Association for Computing Machinery.
- [52] Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. *SIGPLAN Not.*, 38(11):388–402, October 2003.
- [53] Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, pages 289–300, 1993.
- [54] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.
- [55] Michael Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, page 54–61, New York, NY, USA, 2001. Association for Computing Machinery.
- [56] Jiamin Huang, Barzan Mozafari, and Thomas F. Wenisch. Statistical Analysis of Latency Through Semantic Profiling. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 64–79, New York, NY, USA, 2017. Association for Computing Machinery.
- [57] Joseph Izraelevitz, Alex Kogan, and Yossi Lev. Implicit acceleration of critical sections via unsuccessful speculation. *11th ACM SIGPLAN Wkshp. on Transactional Computing*, TRANSACT, 16, 2016.
- [58] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional Memory Architecture and Implementation for IBM System Z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, page 25–36, USA, 2012. IEEE Computer Society.
- [59] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206. IEEE, 2001.
- [60] Richard Johnson, David Pearson, and Keshav Pingali. The Program Structure Tree: Computing Control Regions in Linear Time. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 171–185, New York, NY, USA, 1994. Association for Computing Machinery.
- [61] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. Improving in-memory database index performance with Intel® Transactional Synchronization Extensions. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 476–487. IEEE, 2014.
- [62] Terrance Kelly. Programming Workbench Hand-Over-Hand Locking for Highly Concurrent Collections. https://www.usenix.org/system/files/login/articles/login_fall20_14_kelly.pdf, 2020.
- [63] Andi Kleen. Lock elision in the GNU C library. <https://lwn.net/Articles/534758/>, January 2013.
- [64] Andi Kleen. Scaling Existing Lock-Based Applications with Lock Elision: Lock Elision Enables Existing Lock-Based Programs to Achieve the Performance Benefits of Nonblocking Synchronization and Fine-Grain Locking with Minor Software Engineering Effort. *Queue*, 12(1):20–27, January 2014.
- [65] William Landi and Barbara G. Ryder. Pointer-Induced Aliasing: A Problem Classification. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, page 93–103, New York, NY, USA, 1991. Association for Computing Machinery.
- [66] Johannes Lerch, Johannes Späth, Eric Bodden, and Mira Mezini. Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbounded Access Paths. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, page 619–629. IEEE Press, 2015.
- [67] 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.

- [68] Dmitri Melikyan. Detecting Lock Contention in Go. <https://www.instana.com/blog/detecting-lock-contention-in-go/>, March 2016.
- [69] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, page 314–325, New York, NY, USA, 2008. Association for Computing Machinery.
- [70] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Single Global Lock Semantics in a Weakly Atomic STM. *SIGPLAN Not.*, 43(5):15–26, May 2008.
- [71] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, ZEnterprise EC12, Intel Core, and POWER8. *SIGARCH Comput. Archit. News*, 43(3S):144–157, June 2015.
- [72] Martin Pohlack and Stephan Diestelhorst. From lightweight hardware transactional memory to lightweight lock elision. <https://www.cs.purdue.edu/sss/projects/transact11/papers/Pohlack.pdf>, January 2011.
- [73] Ravi Rajwar and James R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, page 294–305, USA, 2001. IEEE Computer Society.
- [74] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, page 53–64, New York, NY, USA, 2011. Association for Computing Machinery.
- [75] Carl RITSON and Frederick BARNES. An Evaluation of Intel's Restricted Transactional Memory for CPAs. <https://core.ac.uk/download/pdf/18531106.pdf>.
- [76] Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. Transactionalizing legacy code: An experience report using GCC and memcached. *ACM SIGARCH Computer Architecture News*, 42(1):399–412, 2014.
- [77] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '06, page 187–197, New York, NY, USA, 2006. Association for Computing Machinery.
- [78] Michael L. Scott and William N. Scherer III. Scalable queue-based spin locks with timeout. In Michael T. Heath and Andrew Lumsdaine, editors, *Proceedings of the 2001 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'01)*, Snowbird, Utah, USA, June 18-20, 2001, pages 44–52. ACM, 2001.
- [79] Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Robert Geva, Yang Ni, and Adam Welc. Towards Transactional Memory Semantics for C++. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, page 49–58, New York, NY, USA, 2009. Association for Computing Machinery.
- [80] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing Isolation and Ordering in STM. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 78–88, New York, NY, USA, 2007. Association for Computing Machinery.
- [81] Gustavo Sousa and Alexandro Baldassin. FGSCM: A fine-grained approach to transactional lock elision. In *29th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD 2017, Campinas, Brazil, October 17-20, 2017, pages 113–120. IEEE Computer Society, 2017.
- [82] Michael F. Spear, Maged M. Michael, and Christoph von Praun. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, page 275–284, New York, NY, USA, 2008. Association for Computing Machinery.
- [83] 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.
- [84] David Tarjan and Kevin Skadron. Merging path and gshare indexing in perceptron branch prediction. *ACM transactions on architecture and code optimization (TACO)*, 2(3):280–300, 2005.
- [85] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2007.

- [86] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiyang Zhang. Understanding Real-World Concurrency Bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19*, page 865–878, New York, NY, USA, 2019. Association for Computing Machinery.
- [87] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, pages 392–403, 1995.
- [88] Uber. Tally: A Go metrics interface with fast buffered metrics and third party reporters. <https://github.com/uber-go/tally>.
- [89] Filippo Valsorda. Creative foot-shooting with Go RW-Mutex. <https://blog.cloudflare.com/creative-foot-shooting-with-go-rwmutex/>, Oct 2015.
- [90] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT '12*, page 127–136, New York, NY, USA, 2012. Association for Computing Machinery.
- [91] Qingsen Wang, Pengfei Su, Milind Chabbi, and Xu Liu. Lightweight Hardware Transactional Memory Profiling. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, PPOPP '19*, page 186–200, New York, NY, USA, 2019. Association for Computing Machinery.
- [92] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13*, New York, NY, USA, 2013. Association for Computing Machinery.
- [93] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA '08*, page 265–274, New York, NY, USA, 2008. Association for Computing Machinery.
- [94] Zhizhou Zhang, Milind Chabbi, Adam Welc, and Timothy Sherwood. Optimistic Concurrency Control for Real-world Go Programs (Extended Version with Appendix), 2021.
- [95] L. Zheng, X. Liao, H. Jin, and H. Liu. Exploiting the Parallelism Between Conflicting Critical Sections with Partial Reversion. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3443–3457, 2017.
- [96] Lukasz Ziarek, Adam Welc, Ali-Reza Adl-Tabatabai, Vijay Menon, Tatiana Shpeisman, and Suresh Jagannathan. A Uniform Transactional Execution Environment for Java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming, ECOOP '08*, page 129–154, Berlin, Heidelberg, 2008. Springer-Verlag.
- [97] Ferad Zylkyarov, Vladimir Gajinov, Osman Unsal, Adrian Cristal, Eduard Ayguadé, Tim Harris, and Mateo Valero. Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. volume 44, pages 25–34, 04 2009.

Faastlane: Accelerating Function-as-a-Service Workflows

Swaroop Kotni*, Ajay Nayak, Vinod Ganapathy, Arkaprava Basu
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Abstract

In FaaS workflows, a set of functions implement application logic by interacting and exchanging data among themselves. Contemporary FaaS platforms execute each function of a workflow in separate containers. When functions in a workflow interact, the resulting latency slows execution.

Faastlane minimizes function interaction latency by striving to execute functions of a workflow as threads within a single process of a container instance, which eases data sharing via simple load/store instructions. For FaaS workflows that operate on sensitive data, *Faastlane* provides lightweight thread-level isolation domains using Intel Memory Protection Keys (MPK). While threads ease sharing, implementations of languages such as Python and Node.js (widely used in FaaS applications) disallow concurrent execution of threads. *Faastlane* dynamically identifies opportunities for parallelism in FaaS workflows and fork processes (instead of threads) or spawns new container instances to concurrently execute parallel functions of a workflow. We implemented *Faastlane* atop Apache OpenWhisk and show that it accelerates workflow instances by up to 15×, and reduces function interaction latency by up to 99.95% compared to OpenWhisk.

1 Introduction

Function-as-a-Service (FaaS) is emerging as a preferred cloud-based programming paradigm due to its simplicity, client-friendly cost model, and automatic scaling. The unit of computation on a FaaS platform is a developer-provided function. Contemporary FaaS applications typically comprise a set of functions expressed as a *workflow*. A workflow is a directed acyclic graph that specifies the order in which a set of functions must process the input to the application. When an external request such as a web request or a trigger (*e.g.*, timer) arrives for an application, an *instance* of its workflow takes life. AWS Step Functions (ASF) [9], IBM Action Sequences [28], and OpenWhisk Composers [43] enable developers to create and execute such workflows.

*Author is currently affiliated with Microsoft Research India. This work was performed when the author was at the Indian Institute of Science.

FaaS shifts the responsibility of managing compute resources from the developer to the cloud provider. The cloud provider charges the developer (*i.e.*, cloud client) only for the resources (*e.g.*, execution time) used to execute functions in the application (workflow). Scaling is automatic for the developer—as the workload (*i.e.*, number of requests) increases, the provider spawns more instances of the workflow.

In contemporary FaaS offerings, each function, even those that belong to the *same workflow instance*, is executed on a separate container. This setup is ill-suited for many FaaS applications (*e.g.*, image- or text-processing) in which a workflow consists of multiple interacting functions. A key performance bottleneck is *function interaction latency*—the latency of copying *transient* state (*e.g.*, partially-processed images) across functions within a workflow instance. The problem is exacerbated when FaaS platforms limit the size of the directly communicable state across functions. For example, ASF limits the size of arguments that can be passed across functions to 32KB [35]. However, many applications (*e.g.*, image processing) may need to share larger objects [2]. They are forced to pass state across functions of a workflow instance via cloud storage services (*e.g.*, Amazon S3), which typically takes hundreds of milliseconds. Consequently, the function interaction latency could account upto 95% of the execution time of a workflow instance on ASF and OpenWhisk (Figure 6).

Prior works [2, 11, 31, 52] have proposed reducing function interaction latency, but they are far from optimal. First, the proposed software layers to communicate state still incur significant, avoidable overhead. Second, they introduce programming and/or state management complexity. As examples, SAND orchestrates via global message queues for remote communication [2], while Crucial introduces a new programming API for managing distributed shared objects [11].

We observe that if functions of the *same workflow instance* were to execute on threads of a single process in a container, then functions could communicate amongst themselves through load/stores to the shared virtual address space of the encompassing process. The use of load/stores minimizes function interaction latency by avoiding any additional software

overhead and exposes the simplest form of communication possible—no new API is needed. The underlying hardware cache-coherence ensures strong consistency of the shared state at low overhead, without requiring elaborate software management of data consistency, unlike prior works [11, 52].

Faastlane strives to execute functions of a workflow instance on separate threads of a process to minimize function interaction latency. This choice is well-suited in light of a recent study on Azure Functions, which observed that 95% of FaaS workflows consisted of ten or fewer functions [50]. This observation limits the number of threads instantiated in a process. Faastlane retains the auto-scaling benefits of FaaS. Different instances of a workflow spawned in response for each trigger still run on separate containers, possibly on different machines.

While threaded execution of functions simplify state sharing within FaaS workflows, it introduces two new challenges: **① Isolated execution for sensitive data.** Several FaaS use-cases process sensitive data. In such cases, unrestricted sharing of data even within a *workflow instance* leads to privacy concerns. For example, a workflow that runs analytics on health records may contain functions that preprocess the records (*e.g.*, by adding differentially-private noise [19]), and subsequent functions run analytics queries on those records. Workflows are often composed using off-the-shelf functions (*e.g.*, from the AWS Lambda Application Repository [3] or Azure Serverless Community Library [40]). In the example above, raw health records are accessible to a trusted preprocessor function. However, they should not be available to untrusted analytics functions from a library.

Unfortunately, threads share an address space, eschewing the isolated execution of functions within a workflow instance. Thus Faastlane enables lightweight thread-level isolation of sensitive data by leveraging Intel Memory Protection Keys (MPK) [29].¹ MPK allows a group of virtual memory pages (*i.e.*, parts of process address space) to be assigned a specific protection key. Threads can have different protection keys and, thus, different access rights to the same region of the process's address space. The hardware then efficiently enforces the access-rights. Faastlane uses MPK to ensure that functions in a workflow instance, executing on separate threads, have different access rights to different parts of the address space. Simultaneously, it enables efficient sharing of non-sensitive data by placing it in pages shared across the functions.

② Concurrent function execution. Some workflow structures allow many functions within an instance to be executed in parallel. However, most FaaS applications (*e.g.*, 94%) are written in interpreted languages, like Python and Node.js [48], whose popular runtimes disallow concurrent execution of threads by acquiring a global interpreter lock (GIL) [12, 16].

Threaded-execution thus prevents concurrent execution of functions in a workflow instance even if the workflow and the

¹ARM [7] and IBM processors [27] also have MPK-like page-grouping support in their processors. AMD has announced MPK-like feature [6].

underlying hardware admit parallelism. Faastlane addresses this problem via its *adaptive workflow composer*, which analyzes the workflow's structure to identify opportunities for concurrent execution, and forks processes (within the same container) to execute parallel portions of the workflow. Specifically, Faastlane uses threads when the workflow structure dictates sequential execution of functions and processes when there is parallelism. Functions running on separate processes use pipes provided in the multiprocessing module of Python for sharing state. Python pipes internally use shared memory communication. Given that processors with 64-80 cores are commercially available ([46, 47]), while those with 128 cores are on the horizon [45], we expect that containers can be configured with enough vCPUs to allow most workflows to execute entirely within a single container instance. Thus, our key objective is to enable FaaS workflows to leverage efficient communication within a single server.

However, it is also possible for some workflow structures to allow hundreds and thousands of parallel functions. A single container may not have enough vCPUs to fully leverage the available parallelism within a workflow instance. Faastlane's adaptive workflow composer thus judges if the benefit from leveraging parallelism in function execution likely to outweigh the benefits of reduced function interaction latency due to execution of all functions of a workflow instance within one container. If so, the composer falls back on the traditional method of launching multiple containers, each of which would run the number of parallel functions that it can execute concurrently. Functions of a workflow instance executing in separate containers communicate state over the network as happens on contemporary FaaS platforms.

Designing for efficient function interactions also helped us significantly reduce the dollar-cost of executing FaaS applications. On FaaS platforms such as ASF, a developer is charged for **①** the cost of executing functions, **②** the cost of transitioning between nodes of a workflow across containers, and **③** the cost of storage services for transferring large transient states. Faastlane reduces the first two components and eliminates the last one.

To summarize, Faastlane makes the following advances:

- It reduces function interaction latency over OpenWhisk by up to 2307× by executing functions on threads. Consequently, it speeds up a set of example FaaS applications by up to 15×.
- It provides lightweight thread-level isolation using Intel MPK to support applications that process sensitive data.
- It leverages parallelism in workflows by adapting to execute functions as processes, when appropriate, to avoid serialization by GILs in popular FaaS language runtimes.
- Further, if the underlying container cannot fully leverage the available parallelism in a workflow instance, Faastlane falls back on using multiple containers as appropriate.

The source code for Faastlane and the applications are available at <https://github.com/csl-iisc/faastlane>.

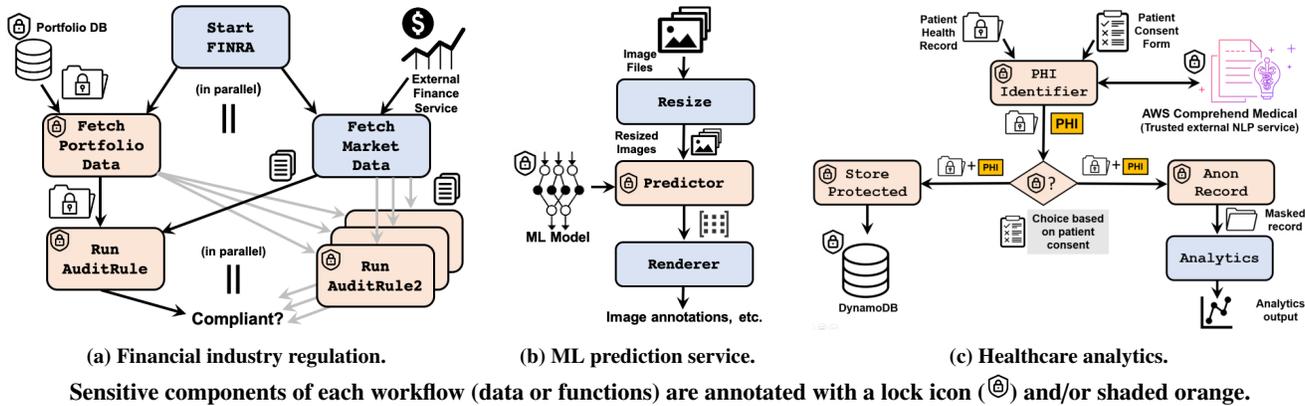


Figure 1: Examples of commercially-important FaaS workflows referenced in Section 2.

2 Function Interaction and State Isolation

We present three commercially-important FaaS workflows that illustrate the importance of minimizing function interaction latency and isolated execution of functions within a workflow instance. We also identify three key design patterns that suffice to express a wide variety of FaaS workflows.

2.1 Financial Industry Regulation

The US Financial Industry Regulatory Authority (FINRA) provides oversight on the operation of broker-dealers to detect malpractices [21]. FINRA requires every broker-dealer to periodically provide it an electronic record of its trades performed during that period. It then validates these trades against market data for about 200 pre-determined rules that check them against a variety of compliance criteria [20].

On average, FINRA validates about half a trillion trades daily [20]. However, this computation only needs to run for some time during the day, and the trade volume to be validated fluctuates daily. The pricing and auto-scaling models of FaaS make FINRA validation an ideal candidate for this platform.

Consider a FaaS workflow to validate a trade against an audit rule (Figure 1a). `StartFINRA` triggers this application’s workflow by invoking two functions. `FetchPortfolioData`, which is invoked on each hedge-fund’s trading portfolio, takes a portfolio ID as input and fetches highly-sensitive trade data, private to each hedge-fund. `FetchMarketData` fetches publicly-available market data based on portfolio type. This function accesses external services to access stock market trade data (e.g., Nasdaq). These functions can run concurrently in a given workflow instance.

The analyzer function (`RunAuditRule`) takes the sensitive output of `FetchPortfolioData` and the non-sensitive information obtained by `FetchMarketData` and validates the trades in each portfolio against the rules. Typically, the size of portfolio data (i.e., the state) shared between the functions runs in KBs, while the market data can be several MBs. Although we have discussed only one checker, FINRA checks compliance against about 200 rules [20]. The checker for each

such rule (e.g., `RunAuditRule2`, etc.) can execute in *parallel* with the other checkers, and each must obtain the outputs of `FetchPortfolioData` and `FetchMarketData`.

To protect sensitive portfolio data from accidental disclosure, `FetchPortfolioData` and `RunAuditRule` simply perform local computations, and do not access external sources. It is also important to ensure that `FetchMarketData` cannot access sensitive data of the other two functions, even though all three functions are part of the same workflow instance of the same application for the same external request.

Current FaaS platforms isolate functions by executing each function in a different container. However, the functions in a workflow may be data-dependent and need to share several MBs of state. Heavy-handed isolation via containers drives up the cost of sharing state and contributes to a significant component of the total workflow execution time (Section 5).

2.2 ML Prediction Service

ML prediction is a computationally-intensive task that takes input data (e.g., images or text), extract features, and provides domain-specific predictions (e.g., detecting objects or predicting sentiments) [17, 34, 52]. Real-world prediction services have real-time response requirements [24]. The workload of an ML prediction service can vary over time and, therefore, can benefit from auto-scaling as provided by FaaS.

Figure 1b presents an example workflow of an ML prediction service for image classification. The workflow has three data-dependent functions, and must, therefore, execute in *sequence*. `Resize` pre-processes an input image and passes the resized image to the main prediction function. `Predictor` consults an ML model and outputs a vector of probabilities that is then post-processed by `Renderer` and rendered suitably e.g., by identifying object boundaries, or suitably labeling objects in an image. The raw image is shared between `Resize` and `Predictor`, and can be 100s-1000s of KB in size, while the output of `Predictor` is in the order of 100s of bytes.

In a typical ML prediction service, the pre-trained ML model is usually the secret sauce, created using sensitive train-

ing data sets and/or at considerable expense. The service provider may not wish to release the model. However, other parts of the workflow may use functions developed by third-parties (e.g., [3, 40]). Here, it is crucial to ensure that `Resize` and `Renderer` do not have access to the state of `Predictor`, which loads the ML model in memory to performs its task.

2.3 Healthcare Analytics

Hospitals generate various health records that can be analyzed to create valuable information for population health management. Applications that process these health records need to meet strict compliance rules (e.g., HIPAA) regarding sensitive data in medical records, including patient details and doctors' notes, which we refer to as protected health information (PHI). Healthcare analytics applications are also a good fit for FaaS pricing and scaling models; AWS Lambda hosts many such applications [41]. Figure 1c depicts an example of healthcare analytics workflow. `PHIIdentifier` takes two inputs: a patient's health record, which may contain unstructured text or images, and a consent form. Using the consent form, the patient can choose to: ① store the medical record securely on a cloud-based service, e.g., DynamoDB in Figure 1c; and ② offer to make available an anonymized version of their medical record to analytics functions downstream. The workflow conditionally executes one or both of the subsequent steps (`StoreProtected` and `AnonRecord`) depending on the patient's consent—we call this design pattern a *choice*.

Both these subsequent steps require identifying PHI data in the unstructured text of the medical record. The `StoreProtected` function segregates PHI data from the medical record, and stores both the PHI data and the stripped medical record securely on a cloud server; the two parts can be combined later when the patient submits a request to retrieve the record. `AnonRecord` anonymizes the medical record by masking out the PHI data identified in the record.

The `PHIIdentifier` function securely communicates with a trusted, external service to identify PHI data in the medical record, e.g., a HIPAA-compliant NLP service like AWS Comprehend Medical [5] or a medical image analyzer like Amazon Rekognition [22]. The medical record and the PHI data are sensitive, and the untrusted `Analytics` function must not access them. The internal state of `PHIIdentifier`, `StoreProtected`, `AnonRecord` must also be protected from `Analytics`. As in the other two examples, this workflow also requires communication of a significant amount of state (e.g., medical records containing images) between functions.

2.4 Function Interaction Patterns in FaaS Workflows

The three workflows discussed thus far illustrate three design patterns for function interaction in FaaS workflows:

- The *parallel* pattern as seen in FINRA, in which functions that are not data-dependent can execute concurrently;
- The *sequence* pattern as seen in the ML prediction service, in which data-dependent functions must execute in sequence;

- The *choice* pattern as seen in Healthcare analytics, in which functions in the workflow are conditionally executed based on the user's input or the output of a function in the workflow.

With these design patterns, we can succinctly encode all constructs of Amazon States Language (ASL) [4, 8]. ASL is the format in which application developers specify workflows for ASF. Thus, it suffices for Faastlane to incorporate support for these three design patterns to express most of the workflows available on contemporary FaaS platforms.

2.5 Threat Model

As illustrated in our examples, functions in a FaaS workflow produce/consume sensitive and non-sensitive data. Even functions that belong to the same workflow instance of a FaaS application should not have unfettered access to each other's state. It is not uncommon for a single FaaS application to use untrusted functions from public repositories (e.g., [3, 40]) along with trusted functions and services that can access sensitive data. As we saw, functions do communicate a substantial amount of state, and the use of any heavy-handed mechanisms for isolation, e.g., executing function within separate containers, inevitably leads to performance bottlenecks.

Our goal is to enable efficient sharing of state across functions in a workflow with just the right amount of isolation. We assume that the cloud provider offers an MPK-based hardware platform and the operating system (OS) used to run FaaS processes includes standard, widely-deployed data-execution prevention features (e.g., $W\oplus X$). Additionally, Faastlane does not change the OS features that aid in defending against microarchitectural side-channel attacks. Best practices to defend against such attacks, such as those developed in Firecracker [1], can also be adapted for use with Faastlane. We implemented Faastlane's mechanisms as extensions to the language runtime (i.e., Python interpreter). We, therefore, assume that the runtime is part of the trusted computing base. Note that we strive to isolate sensitive data and *not* code. Thus, we cannot guard the confidentiality of proprietary functions in FaaS workflows.

3 Design Considerations

Faastlane's design has three objectives. First, we aim to *minimize function interaction latency without sacrificing concurrency in parallel workflows*. It leverages fast communication within a single server (machine) to share state across functions of a single workflow instance, wherever possible. Loads and stores to a shared address space are the lowest-latency options for sharing data. Faastlane thus strives to map functions in a workflow instance to threads sharing a process address space. While threads serve as a vehicle of concurrency in most settings, they do not in this case. Interpreters for popular languages used to write FaaS applications, such as Python and Node.js, use a global interpreter lock (GIL) that prevents concurrent execution of application threads. Faastlane thus has a workflow composer, a static tool that analyzes the workflow

structure, and forks processes instead of threads wherever the workflow allows parallelism. These processes run within the same container and communicate via Python pipes.

The workflow composer is also cognizant of the fact that a single container may not have enough vCPUs to run all functions concurrently (as processes) for workflows that admit massive parallelism. If all functions of an instance of such workflows are packed onto a single container, the performance loss from the lack of concurrency may outweigh the benefits of reducing function interaction latency. Therefore, the composer periodically (*e.g.*, once a day) profiles containers on the FaaS platform to ascertain the available parallelism. Wherever the composer encounters large parallelism in the workflow structure, it determines whether it would be beneficial to deploy instances of that workflow across multiple containers based on measurements from profiling. Each container itself concurrently runs multiple functions as processes, commensurate to that container’s vCPU allocation.

Second, Faastlane aims to *control the sharing of data within a workflow instance* when functions of a workflow instance run as threads. As motivated in Section 2, we discovered important workflows wherein sensitive data must be isolated and shared only with authorized functions. While there is a large literature on in-process isolation techniques [13, 15, 26, 36], recent work [25, 54] has shown that a hardware feature available on Intel server processors, called MPK, offers thread-granularity isolation at low overheads. Faastlane uses Intel MPK to provide thread-granularity memory isolation for FaaS functions that share a virtual address space.

Finally, Faastlane aims to meet the above two objectives *without needing FaaS application writers to modify their functions* and without requiring them to provide more information (than they already do) to FaaS platforms. It is also transparent to the cloud provider, except that Faastlane is most useful when the underlying hardware supports MPK or MPK-like features. Faastlane achieves this goal by designing a static client-side tool, the workflow composer, along with a profiler to achieve its objective of minimizing function interaction latency without sacrificing concurrency. A modified language runtime (here, Python), and the composer, ensure data isolation. Faastlane can be packaged in an enhanced container image. No new API is exposed to FaaS applications, and no additional information is required from developers. This differentiates Faastlane from many prior works (*e.g.*, [11]) that introduce new FaaS programming abstractions (*e.g.*, to specify data sharing). We demonstrate Faastlane’s adoptability by deploying it atop an unmodified OpenWhisk-based FaaS platform on a server equipped with MPK.

4 Implementation of Faastlane

Faastlane needs to accomplish two primary tasks. First, it must spawn threads and/or processes and/or multiple containers to execute functions of a workflow instance for minimizing function interaction latency without sacrificing concurrency that

```

1 def FetchPortfolioData(portfolioName, portfolioType):
2     portfolioData = fetchPrivateData(portfolioName)
3     validateDataFormat(portfolioData)
4     return portfolioData

1 def FetchMarketData(portfolioName, portfolioType):
2     # Accesses a third-party service
3     marketData = yahooFinanceAPI(portfolioType)
4     return marketData

1 def AuditRule([portfolioData, marketData]):
2     # Sample Regulatory rule
3     ruleSatisfied = applyRegulation(
4         portfolioData, marketData)
5     return ruleSatisfied

```

Figure 2: Functions from the FINRA workflow (Figure 1a).

```

1 from FetchPortfolioData import FetchPortfolioData
2 from FetchMarketData import FetchMarketData
3 from AuditRule import AuditRule
4
5 marketData = {}; portfolioData = {}; ruleOut = {}
6
7 def AuditRule_Wrapper(portfolioData, marketData):
8     #Fetch output object(s) from global scope
9     global ruleOut
10    ruleOut = AuditRule([portfolioData, marketData])
11    memset_input(marketData) # zero-out the inputs
12    memset_input(portfolioData)
13
14 def FetchMarketData_Process(portfolio, queue):
15    response = FetchMarketData(portfolio.name,
16                               portfolio.type)
17    queue.put(response)
18
19 def FetchPortfolioData_Process(portfolio, queue):
20    response = FetchPortfolioData(portfolio.name)
21    queue.put(response)
22
23 def PStateWrapper(portfolio):
24    P1 = Process(target = FetchPortfolioData_Process,
25               args = [portfolio, queue1])
26    P2 = Process(target = FetchMarketData_Process,
27               args = [portfolio, queue2])
28    #Execute processes in parallel
29    P1.start(); P2.start()
30    P1.join(); P2.join()
31    #Fetch output object(s) from global scope
32    global marketData, portfolioData
33    #Update output object(s) with queue responses
34    portfolioData = queue1.get()
35    marketData = queue2.get()
36    memset_input(portfolio) # zero-out the inputs.
37
38 def Orchestrator(portfolio):
39    T1 = Thread(target = PStateWrapper,
40              args = [portfolio])
41    T2 = Thread(target = AuditRule_Wrapper,
42              args = [portfolioData, marketData])
43    T1.start(); T1.join()
44    T2.start(); T2.join()
45    return ruleOut

```

Figure 3: Workflow composer output for functions in Figure 2.

exists in the workflow structure. Second, when executing functions as threads, it must ensure the isolation of sensitive state across the functions. Faastlane’s workflow composer, aided by a simple profiler, accomplishes the first task. A modified language runtime and the composer accomplish the second.

4.1 Minimizing Function Interaction Latency

Current FaaS platforms take a workflow description (Figure 1a) and function definitions as inputs (Figure 2), as sepa-

rate entities. They create one container for each of the functions in each executing instance of a workflow. The platform uses the workflow description to suitably route function calls.

Faastlane does not demand any new information from the FaaS application developer or modifications to the code. Faastlane's workflow composer tailors the JSON workflow description and the function code for execution in Faastlane's runtime. The workflow composer first analyzes the workflow structure (JSON description) to identify if it allows for any concurrency in executing the functions of an instance. If the workflow is sequential, the composer packs all functions of the workflow within a unified function called `Orchestrator`. To schedulers on FaaS platforms, the unified workflow provides the *illusion* of an application with a single-function. The entire workflow is thus scheduled for execution in a single container. The workflow composer also creates *function wrappers*, which the `Orchestrator` invokes suitably to reflect the DAG specified in the workflow structure. The `Orchestrator` method's return value is the output of the FaaS application.

Function wrappers explicitly identify the input and output of the function. Wrappers have an associated built-in `start()` method that is implemented in Faastlane's runtime. Invoking a `start()` method spawns a new thread to execute the corresponding function (`join()` denotes the end of the thread). As will be described later in this section, wrappers also implement data isolation using MPK primitives. Each function wrapper begins by reading input from the (shared) process heap and places its output back on the heap. All the other data of a function, not explicitly specified in the wrapper's I/O interface, is considered private to that function and stored in that thread's heap partition (detailed shortly). The `Orchestrator` function specifies the order in which threads are invoked, thereby enforcing the workflow's data sharing policy. The functions of a workflow instance executing as threads minimizes function interaction latency by sharing data using load/stores to the shared process heap.

Language runtimes use a GIL that prevents threads from executing concurrently. This does not present any issues for sequential workflows but leads to loss of concurrency for workflows that contain parallelism. Faastlane launches functions of a workflow instance that can execute in parallel as separate processes instead of threads. Processes can run concurrently on vCPUs of the container deployed on the FaaS platform. These functions communicate via Python pipes.

For example, consider the functions `FetchMarketData` and `FetchPortfolioData` that can run in parallel. The workflow composer creates a parallel-execution wrapper (`PStateWrapper` in Figure 3) that identifies parts of the workflow that can execute concurrently (using the `Process` keyword). The subsequent `start()` method for `P1` and `P2` spawns new processes to execute those portions of the workflow. Each such forked process can itself be a sub-workflow and can further spawn threads or processes as required. The inputs to these functions need not be explicitly copied, as they are

already available in the address space before the `start()` method forks a new process. The output is copied back to the parent process using Python pipes (`queue1` and `queue2`). The `join()` methods in `Orchestrator` serve as barriers that prevent the workflow from progressing unless the corresponding function wrapper thread has completed execution.

Using processes eschews some of the benefits of sharing transient states via loads/stores. Moreover, the language runtime has to be set up in each process, resulting in extra page faults. Typically, functions in FaaS are short-running (<1 sec) [50]). Thus, the overheads due to page faults are non-negligible for these ephemeral processes (detailed in Section 5.2). However, the loss of concurrency with only threads justifies the use of processes for portions of a workflow that admit parallelism.

A container may not have enough vCPUs to run all functions for a parallel section of a workflow concurrently. The number of vCPUs exposed to a container may be smaller than the number of cores in the underlying hardware. Further, FaaS platform providers may not even expose the number of vCPUs on the container it deploys to run a workflow instance. Therefore, Faastlane deploys a simple compute-bound micro-benchmark on the FaaS platform and observes its scalability to infer the number vCPUs in the container deployed by the platform. Such profiling is needed only very infrequently (e.g., once a day) since the number of vCPUs in a container for a given class typically does not change [33].

On encountering a large, parallel workflow structure, the composer spawns multiple containers, each packed with the number of processes equal to the (inferred) number of vCPUs in the container. Networked communication across functions on different containers happens via a REST API [32].

To support cloud hardware that does not offer MPK (or MPK-like) support, the profiler checks for `pku` and `ospke` flags in `/proc/cpuinfo/` to ascertain if the FaaS platform supports MPK. If not, the workflow composer uses only processes to execute functions. This fallback option sacrifices the gains in lowering function interaction latency that threads offer, but adheres to Faastlane's principle of being transparent to both the application writer and the cloud provider.

4.2 Isolation for Sensitive Data

The use of threads for efficient state sharing among functions in a workflow fundamentally conflicts with the goal of protecting sensitive data in situations such as those illustrated in Section 2. Faastlane leverages an existing hardware feature to enforce thread-granularity isolation of data. Note that data isolation is not a concern when functions execute on separate processes. Each process has its own isolated address space, and the state-less nature of FaaS means that they cannot communicate amongst themselves via the local file system, even when they run on the same container.

Faastlane's runtime ensures that each thread gets a private, isolated portion of address space. There is also a shared parti-

tion of the address space that the threads use to communicate the transient state via load/store. Isolation is ensured efficiently using MPK hardware. The memory allocator of the language runtime is extended to ensure that each thread allocates its memory needs from its private partition by default.

Memory protection keys. Intel MPK provides hardware support for each page in a process's address space to be annotated with a 4-bit protection key. Logically, these keys allow us to partition the virtual address space into 16 sets of pages. MPK uses a 32-bit register, called the PKRU register, to specify access rights for each set of pages. Each key is mapped to 2 bits in the PKRU register, that specify access rights of the currently-executing thread to pages in each set. On a memory access, permissions are checked against the PKRU register. Thus, using MPK, it is possible to specify read/write accesses for a set of pages that share the same protection key.

When a process starts, all pages in its address space are assigned a default protection key, and the PKRU value allows both read and write access to all pages. A process assigns a protection key to a page using a system call (`pkey_alloc` in Linux), and the protection key is written into the page-table entry. To specify access rights for pages with that protection key, the process uses the `WRPKRU` instruction, which allows the process to modify the value of the PKRU register from user-space. Since the `WRPKRU` instruction can be executed from user-space, it is important to ensure that it is only invoked from trusted code (called *thread gates*). Faastlane uses binary inspection (as in ERIM [54]) to verify the absence of `WRPKRU` instructions in locations other than thread gates. Faastlane performs this inspection before the workflow is deployed in the FaaS environment. Therefore, it does not incur any run-time performance overheads.

Thread Gates. A thread gate is a sequence of instructions that contains MPK-specific instructions to modify the PKRU register or to assign protection keys to pages. A thread gate is classified as an entry gate or an exit gate, based upon whether it executes at the beginning or end of a thread. Faastlane modifies the Python runtime to implement the instruction sequence corresponding to entry and exit gates in the builtin `start()` and `join()` methods.

An entry gate accomplishes three tasks. First, it attaches a protection key to the thread, using the `pkey_alloc` system call. Second, it communicates this key to the memory manager in Faastlane's Python runtime. The memory manager ensures that all subsequent memory requests are satisfied from pages tagged with the thread's protection key. Finally, the gate uses the `WRPKRU` instruction to write to the PKRU register to ensure that only the current thread has read and write access to pages tagged with that protection key. In effect, this establishes a heap partition accessible only to that thread.

When the thread completes, the exit gate frees the protection key for further use using the `pkey_free` system call. It also zeroes-out the memory region allocated to serve memory for that protection key. Such cleanup allows Faastlane to reuse

protection keys for multiple threads without compromising the isolation guarantees. Without reuse, Faastlane's Python runtime would restrict the number of available protection domains to 16, thereby also restricting the number of functions in the workflow to 15.² One domain is reserved for the parent thread (Orchestrator). The shared heap is accessible to all threads and serves as the shared memory region to which all threads enjoy unfettered read/write access.

Thread Memory Management. Faastlane modifies Python's memory manager to map requests from different threads to different virtual address regions. Faastlane's modifications are packaged as a CPython module (tested for Python3.5). Faastlane modifies the default memory allocator to maintain separate arenas (contiguous regions of virtual address space; typically 256KB) for different threads and ensures that memory requests from one thread are always mapped to the thread's private arenas. After requesting an arena from `mmap`, Faastlane attaches a protection key to the arena using the `pkey_mprotect` system call, effectively reserving the arena for memory requests to the thread owning the protection key.

As discussed, the thread-entry gate associates a protection key for a thread executing a function in the workflow when it starts execution and maps all memory requests from the thread to that private arena. If the arena becomes full, it allocates another arena. When the wrapper finishes, it destroys all the arenas associated with that thread's protection key. The only exception is that of the main thread (Orchestrator), whose arenas are accessible to all threads in the process.

4.3 Putting It All Together

Figure 4 summarizes the lifecycle of a FaaS workflow in Faastlane. The lifecycle starts with the application developer supplying the functions and a workflow description connecting them. The client supplies these entities to Faastlane's workflow composer. The composer analyzes the workflow and produces a unified description using the design patterns discussed in Section 2.4, capturing the available parallelism in the workflow. Based on the available parallelism (if any) and the number of vCPUs in the container (determined via profiling), the workflow is deployed in the FaaS platform.

For workflows without or limited parallelism, the composer provides a single top-level function, called the `Orchestrator`, that encapsulates the entire workflow description. The client supplies this unified workflow description to the FaaS platform, which gives the platform the illusion of a single-function FaaS workflow. The platform schedules the entire FaaS application for execution within a single container. If the workflow contains parallelism that exceeds the estimated number of vCPUs in a container, the composer creates multiple `Orchestrator` functions. Each `Orchestrator` subsumes a sub-workflow with functions that can run concurrently within a single container. Each `Orchestrator` is scheduled on a dif-

²A vast majority of workflows contain fewer than 10 functions [50]. Previous work has also shown ways to virtualize MPK partitions [44].

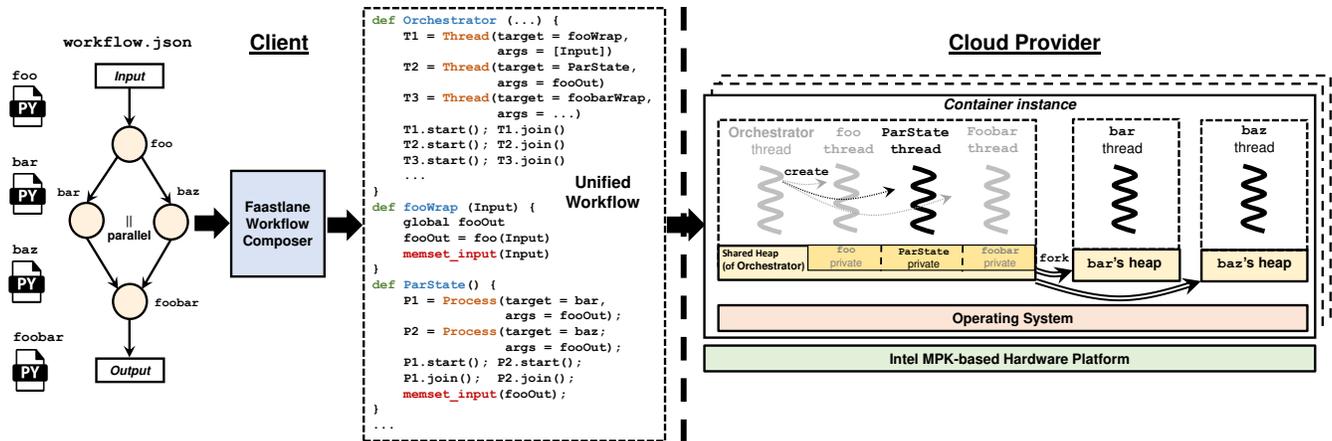


Figure 4: This picture shows a snapshot of the workflow when the functions bar and baz are executing concurrently. The threads shown in grey have either completed execution (foo), are currently paused (Orchestrator) or are yet to be created (foobar). ParState is the wrapper for the bar and baz functions in the parent process. Although not shown in this example, Faastlane allows child processes to be sub-workflows that can recursively spawn threads or processes. If the workflow is highly parallel, the workflow composer creates multiple top-level Orchestrator functions, each of which executes in a different container.

ferent container by the FaaS platform. Faastlane’s runtime component within the container accepts the unified workflow description and decomposes it to identify the functions within the workflow. Based on the Orchestrator, it either starts threads (one per function) or forks processes to execute functions of the workflow. When functions execute as threads, each thread executes instructions to identify a partition of the process heap accessible only to that thread (using MPK primitives). It uses this partition to store data private to the function. The functions running on the thread can share state via load/stores to a designated shared heap for the process. When a thread completes, the function’s output is made available (via the Orchestrator method) only to the functions in the workflow that consume that output. When Faastlane forks a process, it passes only the output of the last method from the parent process to the Orchestrator method of the child process via a Python pipe. For large parallel workflows, functions in different containers communicate over the network stack as in contemporary FaaS platforms.

Faastlane does not impact auto-scaling in FaaS. Different instances of a given workflow spawned in response to a trigger are scheduled on different containers, as is typical.

5 Evaluation

We evaluate Faastlane against four real-world applications that broadly capture popular FaaS function interaction patterns [39]. Besides the applications in Section 2, we include Sentiment analysis, which is another real-world use case [49]. Sentiment analysis evaluates user reviews for different products of a company. Its workflow contains two choice states. The first choice state chooses an analysis model based on the product. The second choice state publishes to a database versus a message queue (for manual analysis of negative reviews) based on review sentiments. Together, these applications en-

Processor	2 × Intel(R) Xeon(R) Gold 6140 CPU
No. of cores	36 (2 × 18)
DRAM	384 GB, 2666 MHz
LLC Cache	24 MB
Linux Kernel	v. 4.19.90
Docker	19.03.5, build 633a0ea838

Table 1: Experimental Setup.

capsulate all the patterns discussed in Section 2.4. Table 1 shows the configuration of our experimental system, including the software stack. We measure how Faastlane improves function interaction latency, end-to-end latency, throughput, and reduces dollar-cost. We perform the experiments on a single machine where Faastlane can pack all functions of a workflow instance within a single container. However, in Section 5.4 we further show how Faastlane scales up to use multiple containers in face of large parallelism within a workflow structure.

We compare Faastlane’s performance against ASF, OpenWhisk and SAND [2]. SAND aims to reduce function interaction latency. It executes functions in a workflow as separate processes within a container and shares state using a hierarchical message queue. Our performance experiments with OpenWhisk, SAND and Faastlane were run on our local hardware platform, while ASF experiments were run on the AWS cloud infrastructure. We cannot control the AWS cloud hardware platform. Thus, the ASF measurements are not necessarily directly comparable to those of the other platforms.

5.1 Function Interaction Latency

Figure 5 reports the function interaction latency of the four applications under different FaaS platforms. The applications are as described in Section 2, with the only difference that the FINRA application checks portfolio data against 50 audit rules. We run each application at least 100 times and report the median. We observe that function interaction latency is

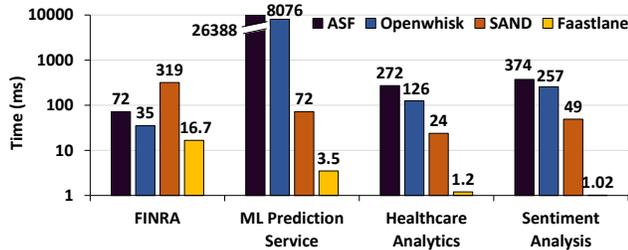


Figure 5: Median function interaction latency (in milliseconds)

lowest with Faastlane across all four applications, each with different function interaction patterns. Faastlane reduces the latency over its *closest* competitor by 52.3%, 95.1%, 95% and 98% for FINRA, ML prediction service, Healthcare analytics and Sentiment analysis, respectively.

SAND reduces the function interaction latency compared to current FaaS platforms, except for FINRA. FINRA runs 50 audit rules in parallel, where SAND’s hierarchical messaging queues serialize communication from concurrently executing functions. Consequently, SAND increases the function interaction latency to 319ms from 35ms on OpenWhisk. However, Faastlane reduces it to 16.7ms. For ML prediction service, SAND reduces the interaction latency to 72ms from thousands of milliseconds on OpenWhisk while Faastlane almost eliminates it. Similarly, for the remaining apps, the interaction latency is a mere 1-1.2ms with Faastlane.

Note that the ML prediction service has very high function interaction latencies (in thousands of milliseconds) on both ASF and OpenWhisk. This is because the ML prediction service transmits the largest state across functions—an image of about 1.6MB size from the Resize function to the Predictor function (Figure 1b). However, on ASF and OpenWhisk, functions in a workflow can directly transmit a maximum state of size 32KB [35] and 1MB, respectively. Consequently, the ML prediction service is forced to fall back upon a slow storage service like Amazon S3 for state sharing on these platforms. The other applications transmits tens of KBs of state across functions in a workflow (for the payloads we used).

Though Faastlane offers the lowest function interaction latency for FINRA, the latency of 16.7ms is higher than the latency observed for the other applications even if it transmits only tens of KBs of state across functions. This behavior is because FINRA’s workflow invokes 50 functions in parallel to audit the portfolios. Faastlane executes these functions as separate processes where the state is shared via Python pipes, instead of loads/stores within a process address space. While this helps end-to-end latency by leveraging concurrency (Section 5.2), the cost of sharing state increases compared to sharing state across threads of the same process.

5.2 End-to-End Application Performance

Latency. The end-to-end execution latency of an application’s workflow is the time that elapses between the start of the first function of the workflow and the completion of the final function in the workflow. We measure end-to-end latency by

executing each application at least 100 times and report both the median and tail (99%-ile) values.

We dissect the end-to-end latency as follows:

- ① The time spent in external service requests. An external service request is one that requires communication outside the FaaS platform. For example, the healthcare analytics application uses the AWS Comprehend Medical service to identify PHI in health records (Section 2.3). This component depends on several factors, including time over the network and the latency of processing the request at the external entity.
- ② The compute time, which is the time during which the workflow functions execute on the processor. It does not include the time spent on external service requests.
- ③ The function interaction latency on the FaaS platform.

Figure 6 presents the end-to-end latency broken down into the above components. For each application, we report median and 99%-ile latency on ASF, OpenWhisk, SAND, and Faastlane. The total height of each bar represents the end-to-end latency, and the stacks in the bar show the breakdown. In Figure 6a, we first note that the compute time is significantly more on ASF than on other platforms. While the FINRA application has 50 parallel functions, ASF never runs more than 10-12 of them concurrently. On other platforms, the compute time is much shorter because all the functions execute in parallel. Most of the time is spent on the external service request for retrieving the trade data. Overall, we find that Faastlane improves end-to-end latency by 40.5%, 23.7% over OpenWhisk and SAND, respectively.

In ML prediction service (Figure 6b), OpenWhisk’s and ASF’s end-to-end execution latency are dominated by the function interaction latency, as discussed earlier. In comparison, SAND significantly reduces function interaction latency, while Faastlane practically eliminates it. Consequently, even in comparison to SAND, Faastlane improves end-to-end latency by 22.3%. Over OpenWhisk, the improvement due to Faastlane is 15×. Note that the compute time on OpenWhisk is shorter (by about 100ms) than that on Faastlane and SAND, although we ran OpenWhisk, Faastlane, and SAND on the same hardware platform. This difference is because, on Faastlane and SAND, the ML prediction service application transmits data between functions using JSON (de)serialization methods, contributing to the compute time. In contrast, on OpenWhisk, the application is forced to use S3, and the application does not use JSON when communicating with S3.

In Healthcare analytics (Figure 6c), the time spent on external service request (AWS Comprehend Medical) is significant. The response latency from AWS Comprehend Medical depends on external factors and is orthogonal to our system design. We observed that the latency is much lower on ASF but similar on the FaaS platforms that were executed on our local hardware. In comparison to OpenWhisk and SAND, Faastlane improves end-to-end latency by 9.9% and 10.4%, respectively. Faastlane improves end-to-end latency of Sen-

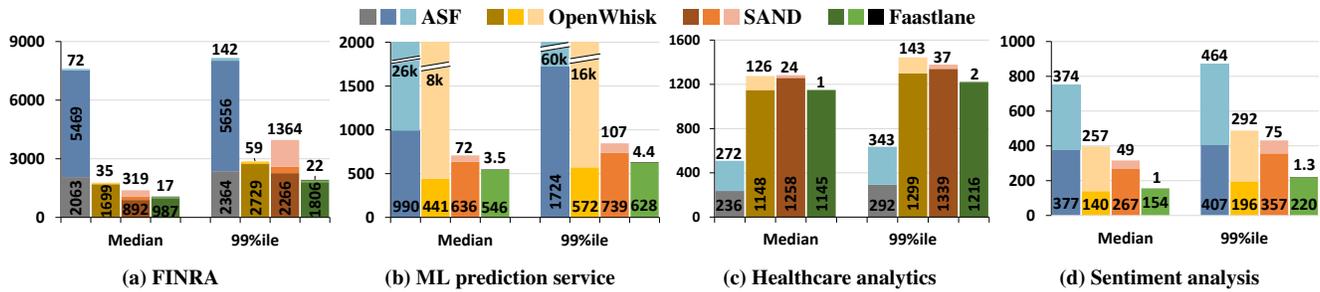


Figure 6: Application end-to-end latency (in ms). Legend shows external service time, compute time and function interaction latency for each platform in that order. Note that (b), (d) do not have external services and (c) has negligible compute time (2-4ms).

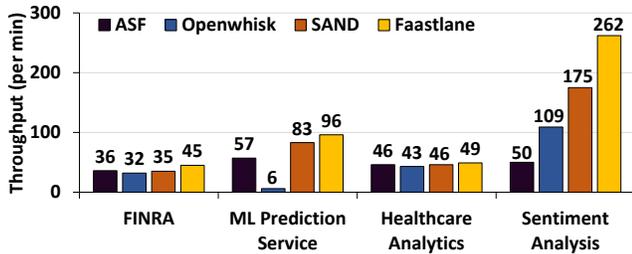


Figure 7: Measuring application throughput.

timent analysis (Figure 6d) by 61% over OpenWhisk and 51% over SAND. The tail (99%-ile) latency trends (Figure 6), mirror the above observations. In short, Faastlane provides the least end-to-end latency across platforms that are directly comparable, *i.e.*, run on the same hardware.

Throughput. Figure 7 shows the application throughput, measured as the number of application requests serviced per minute on different FaaS platforms. We use the time between the invocation of the first request and the response receipt for the last request. We repeat this experiment over several requests to measure throughput as the number of requests-per-minute. Unlike end-to-end execution latency, throughput measurement also accounts for initialization (*e.g.*, spawning containers), and post-processing time. Note that to compare throughput, we also need to ensure that all configurations run on the same hardware. The total number of cores on the hardware platform (and therefore, the available parallelism) has a *first-order* effect on throughput, unlike latency. While we present numbers from ASF for completeness, we forewarn the readers against directly comparing ASF's throughput numbers (run on hardware not in our control), with the numbers obtained on other FaaS platforms executed locally.

Figure 7 shows the throughput observed for each of our applications on ASF, OpenWhisk, SAND and Faastlane. In general, we observe that throughput follows the trends in the end-to-end latency. Faastlane provides the best throughput across all applications. For the FINRA application, Faastlane improves throughput by 28.6% over its nearest competitor (SAND). For ML prediction service and Sentiment analysis, Faastlane improves throughput by 15.6%, 49.7% over SAND and by 16 \times , 2.4 \times over OpenWhisk, respectively. For the healthcare analytics application, Faastlane provides a modest throughput improvement of 6% over SAND.

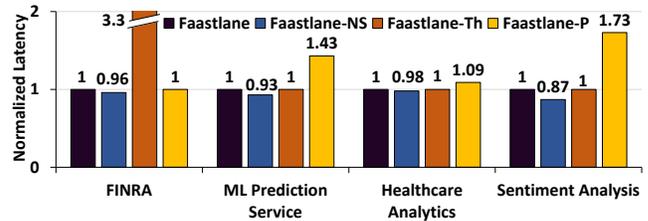


Figure 8: Analysis of the cost of isolation in Faastlane.

Cost of Isolation. For a deeper understanding of the design choices made in Faastlane, we create three variants of Faastlane. First, we turn off thread-level memory isolation in Faastlane by disabling its MPK-enabled memory manager and the zeroing-out of function's state in the shared domain (Faastlane-NS). The difference in performance between Faastlane-NS and Faastlane quantifies the performance cost of data isolation using MPK. Second, we constrain Faastlane to only use threads, and not a mix of processes and threads based on the workflow structure (Faastlane-Th). The performance difference between Faastlane-Th and Faastlane quantifies the usefulness of adapting between processes and threads. Third, we constrain Faastlane to use *only* processes, and share state via Python pipes (Faastlane-P). The performance difference between Faastlane-P and Faastlane quantifies the importance of employing threads when possible.

Figure 8 shows the normalized median end-to-end latency for Faastlane, Faastlane-NS, Faastlane-Th and Faastlane-P for all four applications. The height of each bar is normalized to latency under Faastlane. Comparing Faastlane-NS and Faastlane, we observe an increase of 1.9-14.9% in the end-to-end latency in Faastlane, which denotes the cost of MPK-based isolation mechanism. We argue that this performance cost is a reasonable price to pay for data isolation.

Next, we observe that the latency for Faastlane-Th is 3.3 \times of the Faastlane for FINRA. Recall that FINRA has 50 parallel functions in its workflow. Since the threads cannot execute concurrently in our Python implementation (due to the GIL), Faastlane-Th cannot execute these parallel functions concurrently. In contrast, Faastlane employs processes instead of threads and can leverage the underlying hardware resources for concurrency. For the rest of the applications, there is no difference between Faastlane-Th and Faastlane as they do not

Application	AWS Step Functions (ASF)				Faastlane Lambda
	Lambda	Step	Storage	Total	
FINRA	31.17	1325	0	1356.17	10.2
ML Prediction Service	21.75	125	27.85	174.6	11.87
Healthcare Analytics	1.65	150	0	151.65	0.83
Sentiment Analysis	1.85	175	0	176.85	1.03

Table 2: Cost (in USD) per 1 million application requests

contain any parallel functions. This experiment shows the importance of analyzing the workflow structure and using processes instead of threads in the presence of GIL.

Finally, the latency for Faastlane-P is 9-73% higher than Faastlane in applications except FINRA. Lifetimes of FaaS functions are short, *e.g.*, 50% of the functions execute in less than 1s [50]. Thus, processes running these functions in Faastlane-P are ephemeral and we found the page-fault latency in setting up language runtime in each process dominates their execution times. Faastlane reduces page-faults by 1.43-3.95 \times using threads over processes. For FINRA, Faastlane uses processes to leverage parallelism. Thus, there is no difference between Faastlane-P and Faastlane.

5.3 Dollar-cost of Application Execution

We now discuss the dollar-cost implications to developers for executing applications on Faastlane versus atop ASF. The cost of executing applications on ASF includes: ① *Lambda* costs, which is the cost of executing the functions of a workflow, ② *Step* costs, which is the cost of transitioning from one node to another in an ASF workflow across containers, and ③ *Storage* costs, for transferring large transient state [53]. We noticed that execution times of functions on ASF varies run-to-run. We used the smallest execution time to estimate the least cost of running an application on ASF.

Unlike with ASF, the dollar-cost of running on Faastlane *only* includes the *Lambda* cost. The *Step* cost of transitioning across containers is zero because Faastlane executes an entire workflow instance within a container. Faastlane does not need a storage service to transfer state and therefore incurs no *Storage* cost. To measure the *Lambda* cost on Faastlane, we deployed applications with Faastlane-enhanced virtual environments on AWS Lambda. We disabled MPK instructions because AWS machines may not have MPK-enabled today. However, we adjusted the billed-duration and maximum memory of each application with MPK overheads (1.9-14.9%). We exclude the cost of any external service that a function uses for its execution (*e.g.*, AWS Comprehend Medical).

Table 2 reports the total cost incurred for each application on ASF and Faastlane. Faastlane executes applications only at 0.6-6.8% of the cost of ASF. Faastlane eliminates expensive step costs and storage costs by designing for efficient function interactions. Despite running on containers with larger memory, Faastlane reduces Lambda costs, partly due to the reduction in runtime because of efficient function interactions and partly due to the current AWS Lambda pricing model.

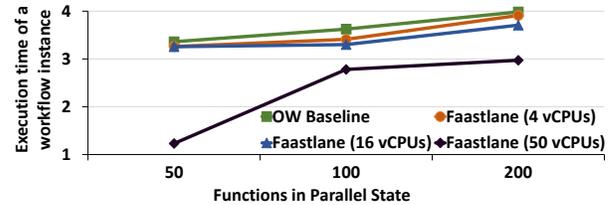


Figure 9: Scalability for FaaS Applications with parallelism.

5.4 Scalability with Multiple Containers

All experiments thus far were limited to one server, and the underlying container had sufficiently many vCPUs to run all parallel functions in applications like FINRA concurrently. We now explore how Faastlane would scale if the underlying container had a limited number of vCPUs and when multiple servers are to be deployed to cater to the parallelism available within a single workflow instance. For this purpose, we scaled up the number of functions in the parallel step in FINRA’s workflow from 50 to 200. We also varied the number of vCPUs in each container from 4 to 50.

When Faastlane detects that the parallelism in the workflow exceeds the parallelism serviceable by the vCPUs of a single container, it spawns multiple containers, possibly scheduled on multiple servers. Each container is packed with a number of processes commensurate with the vCPUs allotted to it.

Figure 9 shows how Faastlane performs under such circumstances vis-a-vis OpenWhisk baseline that runs every function in separate containers. The y-axis reports the average execution time of a workflow instance as the number of parallel functions are scaled (x-axis). We observe that when the number of vCPUs per container is limited, Faastlane offers limited uplift (2.46%). This is expected since Faastlane falls back on network-based communication across containers when spawning multiple containers. As we increase the number of vCPUs per container, Faastlane launches fewer containers, and performs much better. At 50 vCPUs per container, Faastlane speeds up by 2.73 \times over the baseline when there are 50 parallel functions. Even when there are 200 functions, Faastlane reduces execution time by 25.3%. In short, Faastlane is most effective when it can leverage fast communication within a container, particularly for sequential workflows and those with limited parallelism. However, even when faced with large parallelism in workflows, coupled with limited parallelism in underlying containers, Faastlane scales at least as well as the current offerings.

6 Related Work

Table 3 compares Faastlane against closely related systems using three criteria—function interaction latency, ability to support unmodified FaaS workflows (*e.g.*, those written for commodity FaaS offerings such as OpenWhisk and ASF), and whether the solution supports function interaction across machines. Commercial FaaS solutions satisfy the latter two criteria, but have high function interaction latencies [10, 23].

	Function Interaction Latency	Unmodified Application	Interaction Across Machines
OpenWhisk/ASF	High	✓	✓
SAND [2]	Medium	✓	✓
SONIC [37]	Medium	✓	✓
Crucial [11]	Medium	✗	✓
Cloudburst [52]	Medium	✗	✓
Faasm [51]	Low	✗	✗
Nightcore [30]	Low	✗	✗
Faastlane	Low	✓	✓

Table 3: Comparing Faastlane with related work.

SAND [2] reduces function interaction latency via hierarchical message queues, while allowing unmodified FaaS applications and workflows spanning multiple machines. However, as our evaluation showed, Faastlane’s approach of using memory load/store instructions reduces function interaction latency even further. SONIC [37] aims to reduce function interaction latency by jointly optimizing how data is shared (e.g., by sharing files within a VM or copying files across VMs) and the placement of functions in a workflow. However, SONIC uses files as the core data-sharing mechanism, leaving room for latency reduction in several cases.

Crucial [11] improves state-sharing in highly parallelizable functions using distributed shared objects (DSOs). DSOs are implemented atop a modified Infinispan in-memory data grid. Unfortunately, Crucial’s approach is not compatible with unmodified FaaS applications. Cloudburst [52] focuses on distributed consistency of shared data using Anna key-value stores [56]; other systems, like Pocket [31], use specialized distributed data stores for transient data. These systems improve function interaction latency, but leave significant headroom for improvement because of the overheads of accessing the remote data store.

While all the aforementioned systems support distributed FaaS applications, in which interacting functions execute on different machines, Faasm [51] and Nightcore [30] aim to reduce function interaction latency by executing workflows on a single machine. Faasm [51] uses threads and shared memory with software-fault isolation [55] (more heavy-weight) to provide thread-private memory partitions. Unfortunately, Faasm requires custom-written FaaS applications that are then compiled down to WebAssembly for execution. Nightcore [30] accelerates stateless microservices by co-locating them on the same physical machine and using shared memory to enable efficient data sharing. However, the FaaS applications must be modified to use Nightcore’s libraries to avail of these benefits. Faastlane’s function interaction latencies are comparable to Faasm and Nightcore, but it additionally supports unmodified FaaS workflows and distributed FaaS applications.

Aside from these systems that share Faastlane’s goal of reducing function interaction latency in FaaS workflows, there is prior work on reducing the memory footprint and performance overhead of other aspects of FaaS. Chief among these are methods that aim to reduce the memory footprint and performance overheads of containers by creating stripped-down containers [42], checkpointing [18], language-level iso-

lation [14], or create bare-bones VMs tailored to the application loaded in the VM [1, 38]. Faastlane’s lightweight sandboxing approach can be used in conjunction with prior work to improve the overall performance of FaaS applications.

Faastlane uses MPK to offer lightweight memory isolation between threads when they are used in the workflow. Recent works introduce new OS-level primitives for efficient in-process isolation [13, 15, 26, 36]. However, the overheads are still substantial compared to MPK which offers user-level instructions to switch memory domains. In contrast, the other solutions add new system calls, or introduce non-negligible runtime overheads. Prior studies have aimed to provide intra-process isolation using MPK. We borrow the ideas of thread gates and binary inspection from ERIM [54]. While Faastlane creates in-process thread-private memory domains, Hodor [25] creates protected libraries using MPK-backed protection domains. MPK offers 16 domains, which is sufficient for most current FaaS applications [50]. For larger workflows, Faastlane reuses thread domains wherever possible (Section 4.2). Libmpk [44] offers a software abstraction of MPK to create a larger number of domains. However, it adds non-negligible software overheads.

7 Conclusion

Developers structure FaaS applications as workflows consisting of functions that interact by passing transient state between each other. Commercial FaaS platforms execute functions of a workflow instance in different containers. The transient state must be copied across these containers or transmitted via cloud-based storage services, both of which contribute to the latency of function interaction.

Faastlane reduces function interaction latency by sharing data via memory load/store instructions. To accomplish this goal, it strives to execute functions of a workflow instance as threads within a shared virtual address space. However, this approach raises new challenges associated with isolating sensitive data and availing the parallelism afforded by the underlying hardware. Faastlane’s novel design uses Intel MPK for lightweight thread-level isolation and an adaptive mixture of threads and processes to leverage hardware parallelism. Our experiments show that Faastlane outperforms commercial FaaS offerings and also recent research systems designed to reduce function interaction latency.

Acknowledgments

We are grateful to the anonymous reviewers for their thoughtful feedback on the work. Thanks to Onur Mutlu for shepherding the paper. This work was supported in part by a Ramanujan Fellowship from the Government of India, a Young Investigator fellowship from Pratiksha Trust, and research gifts from Intel Labs and VMware Inc.

References

- [1] 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)*, February 2020.
- [2] 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 *USENIX Annual Technical Conference*, 2018.
- [3] Amazon. AWS Serverless Application Repository—Discover, deploy, and publish serverless applications. <https://aws.amazon.com/serverless/serverlessrepo/>.
- [4] Amazon States Language—AWS Step Functions Developer Guide. <https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html>.
- [5] Amazon Comprehend Medical—extract information from unstructured medical text accurately and quickly. <https://aws.amazon.com/comprehend/medical/>.
- [6] Memory Protection Keys on AMD processors. <https://www.anandtech.com/show/16214/amd-zen-3-ryzen-deep-dive-review-5950x-5900x-5800x-and-5700x-tested/6>.
- [7] ARM. ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition, 2018.
- [8] Amazon States Language Specifications. <https://states-language.net/spec.html>.
- [9] AWS Step Functions—build distributed applications using visual workflows. <https://aws.amazon.com/step-functions/>.
- [10] Daniel Barcelona-Pons, Pedro García-López, Álvaro Ruiz, Amanda Gómez-Gómez, Gerard París, and Marc Sánchez-Artigas. Faas orchestration of parallel workloads. In *Proceedings of the 5th International Workshop on Serverless Computing*, 2019.
- [11] Daniel Barcelona-Pons, Marc Sanchez-Artigas, Gerard Paris, Pierre Sutra, and Pedro Garcia-Lopez. On the FaaS Track: Building stateful distributed applications with serverless architectures. In *ACM Middleware Conference*, 2019.
- [12] David Beazly. Inside the Python GIL. In *Python Concurrency and Distributed Computing Workshop*, May 2009. <http://www.dabeaz.com/python/GIL.pdf>.
- [13] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.
- [14] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.
- [15] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *IEEE Symposium on Security and Privacy*, 2016.
- [16] Python—Global Interpreter Lock. <https://wiki.python.org/moin/GlobalInterpreterLock>.
- [17] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *USENIX Symposium on Networked Systems Design and Implementation*, 2017.
- [18] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyst: 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*, 2020.
- [19] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3–4):211–407, 2014.
- [20] FINRA adopts AWS to perform 500 billion validation checks daily. <https://aws.amazon.com/solutions/case-studies/finra-data-validation/>.
- [21] United States Financial Industry Regulatory Authority. <https://www.finra.org/>.
- [22] Sarah Gabelman. How to use Amazon Rekognition and Amazon Comprehend Medical to get the most out of medical imaging data in research, September 2019. <https://aws.amazon.com/blogs/apn/how-to-use-amazon-rekognition-and-amazon-comprehend-medical-to-get-the-most-out-of-medical-imaging-data-in-research/>.
- [23] Pedro Garcia-Lopez, Marc Sanchez-Artigas, Gerard Paris, Daniel Barcelona-Pons, Alvaro Ruiz Ollobarren, and David Arroyo Pinto. Comparison of FaaS Orchestration Systems. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018.

- [24] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dymtro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yanqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiadong Wang. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *IEEE International Symposium on High Performance Computer Architecture*, 2018.
- [25] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, July 2019.
- [26] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing least privilege memory views for multithreaded applications. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2016.
- [27] IBM. Power ISATM version 3.0 b, 2017.
- [28] IBM Cloud Functions—Creating Sequences. <https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-sequences>.
- [29] Intel. Intel-64 and IA-32 architectures software developer’s manual, 2018.
- [30] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.
- [31] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.
- [32] API Gateway for AWS Lambda. <https://docs.aws.amazon.com/lambda/latest/dg/services-apigateway.html>.
- [33] Container Configuration in AWS Lambda. https://docs.amazonaws.cn/en_us/lambda/latest/dg/configuration-memory.html.
- [34] Yunseong Lee, Alberto Scolari, Byoung-Gon Chun, Marco D. Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the black box of machine-learning prediction serving systems. In *USENIX Symposium on Operating Systems Design and Implementation*, 2018.
- [35] AWS Step Function limits. <https://docs.aws.amazon.com/step-functions/latest/dg/limits.html>.
- [36] 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 *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, 2016.
- [37] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.
- [38] 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*, 2017.
- [39] Microsoft. Azure durable popular application patterns. <https://docs.microsoft.com/en-in/azure/azure-functions/durable/durable-functions-overview?tabs=csharp#application-patterns>.
- [40] Microsoft. Azure serverless community library. <https://serverlesslibrary.net/>.
- [41] Chris Munns. Powering HIPAA-compliant workloads using AWS Serverless technologies. In *AWS Compute Blog*, July 2018. <https://aws.amazon.com/blogs/compute/powering-hipaa-compliant-workloads-using-aws-serverless-technologies/>.
- [42] 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)*, 2018.
- [43] Apache OpenWhisk Composer—A high-level programming model in JavaScript for composing serverless functions. <https://github.com/apache/openwhisk-composer>.
- [44] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *USENIX Annual Technical Conference*, 2019.
- [45] 128 Core Processor. <https://www.crn.com/news/components-peripherals/ampere-s-new-128-core-altra-cpu-targets-intel-amd-in-the-cloud>.
- [46] 80 Core Processor. <https://amperecomputing.com/ampere-altra-industrys-first-80-core-server-processor-unveiled/>.

- [47] AMD Epyc 7002 series. <https://www.amd.com/en/processors/epyc-7002-series>.
- [48] Ran Ribenzaft. What AWS Lambda's performance stats reveal, April 2019. <https://thenewstack.io/what-aws-lambdas-performance-stats-reveal/>.
- [49] Serverless data processing with AWS step functions. <https://medium.com/weareservian/serverless-data-processing-with-aws-step-functions-an-example-6876e9bea4c0>.
- [50] Mohammad Shahrads, 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)*, 2020.
- [51] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.
- [52] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M. Faleiro, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful Functions-as-a-Service. In *International Conference on Very Large Data Bases*, 2020.
- [53] AWS Step Functions pricing. <https://aws.amazon.com/s3/pricing/>.
- [54] 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 *USENIX Security Symposium*, 2019. Tool available at: <https://gitlab.mpi-sws.org/vahldiek/erim/-/tree/master/>.
- [55] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *ACM Symposium on Operating Systems Principles*, 1993.
- [56] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph M. Hellerstein. Anna: A KVS for Any Scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018.

SONIC: Application-aware Data Passing for Chained Serverless Applications

Ashraf Mahgoub
Purdue University

Karthick Shankar
Carnegie Mellon University

Subrata Mitra
Adobe Research

Ana Klimovic
ETH Zurich

Somali Chaterji
Purdue University

Saurabh Bagchi
Purdue University

Abstract

Data analytics applications are increasingly leveraging serverless execution environments for their ease-of-use and pay-as-you-go billing. The structure of such applications is usually composed of multiple functions that are chained together to form a workflow. The current approach of exchanging intermediate (ephemeral) data between functions is through a remote storage (such as S3), which introduces significant performance overhead. We compare three data-passing methods, which we call *VM-Storage*, *Direct-Passing*, and state-of-practice *Remote-Storage*. Crucially, we show that no single data-passing method prevails under all scenarios and the optimal choice depends on dynamic factors such as the size of input data, the size of intermediate data, the application’s degree of parallelism, and network bandwidth. We propose SONIC, a data-passing manager that optimizes application performance and cost, by transparently selecting the optimal data-passing method for each edge of a serverless workflow DAG and implementing communication-aware function placement. SONIC monitors application parameters and uses simple regression models to adapt its hybrid data passing accordingly. We integrate SONIC with OpenLambda and evaluate the system on Amazon EC2 with three analytics applications, popular in the serverless environment. SONIC provides lower latency (raw performance) and higher performance/\$ across diverse conditions, compared to four baselines: SAND, vanilla OpenLambda, OpenLambda with Pocket, and AWS Lambda.

1 Introduction

Serverless computing platforms provide on-demand scalability and fine-grained resource allocation. In this computing model, cloud providers run the servers and manage all administrative tasks (e.g., scaling, capacity planning, etc.), while users focus on the application logic. Due to its elasticity and ease-of-use advantages, serverless computing is becoming increasingly popular for advanced workflows such as data processing pipelines [35, 53], machine learning pipelines [11, 55], and video analytics [5, 22, 64]. Major cloud providers recently

introduced serverless workflow services such as AWS Step Functions [4], Azure Durable Functions [45], and Google Cloud Composer [24], which provide easier design and orchestration for serverless workflow applications. Serverless workflows are composed of a sequence of execution stages, which can be represented as a directed acyclic graph (DAG) [17, 53]. DAG nodes correspond to serverless functions (or λ s¹) and edges represent the flow of data between dependent λ s (e.g., our video analytics application DAG is shown in Fig. 1).

Exchanging intermediate data between serverless functions is a major challenge in serverless workflows [34, 36, 47]. By design, IP addresses and port numbers of individual λ s are not exposed to users, making direct point-to-point communication difficult. Moreover, serverless platforms provide no guarantees for the overlap in time between the executions of the parent (sending) and child (receiving) functions. Hence, the state-of-practice technique for data passing between serverless functions is saving and loading data through remote storage (e.g., S3). Although passing intermediate data through remote storage has the benefit of cleanly separating compute and storage resources, it adds significant performance overheads, especially for data-intensive applications [47]. For example, Pu *et al.* [53] show that running the CloudSort benchmark with 100TB of data on AWS Lambda with S3 remote storage can be up to 500 \times slower than running on a cluster of VMs. Our own experiment with a machine learning pipeline that has a simple linear workflow shows that passing data through remote storage takes over 75% of the computation time (Fig. 2, fanout = 1). Previous approaches reduce this overhead by implementing exchange operators optimized for object storage [47, 52], replacing disk-based object storage (e.g., S3) with memory-based storage (e.g., ElastiCache Redis), or combining different storage media (e.g., DRAM, SSD, NVMe) to match application needs [12, 37, 53]. However, these approaches still require passing data over the network multiple times, adding latency. Moreover, in-memory storage services are much more expensive than disk-based stor-

¹For simplicity, we denote a serverless function as lambda or λ .

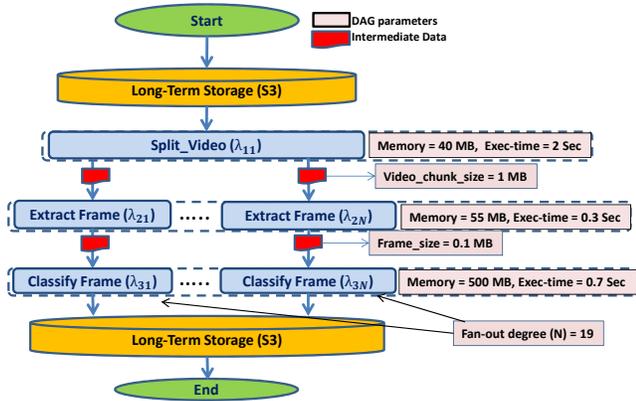


Figure 1: DAG overview (DAG definition provided by the user and our profiled DAG parameters) for Video Analytics application.

age (e.g., ElastiCache Redis costs 700× more than S3 per GB [53]).

We show how cloud providers can optimize data exchange between chained functions in a serverless DAG workflow with communication-aware placement of lambdas. For instance, the cloud provider can leverage data locality by scheduling the sending and receiving functions on the same VM, while preserving local disk state between the two invocations. This data passing mechanism, which we refer to as *VM-Storage*, minimizes data exchange latency but imposes constraints on where the lambdas can be scheduled. Alternatively, the cloud provider can enable data exchange between functions on different VMs by directly copying intermediate data between the VMs that host the sending and receiving lambdas. This data passing mechanism, which we refer to as *Direct-Passing*, requires one copy of the intermediate data elements, serving as a middle ground between *VM-Storage* (which requires no data copying) and *Remote storage* (which requires two copies of the data). Each data passing mechanism provides a trade-off between latency, cost, scalability, and scheduling flexibility. Crucially, we find that no single mechanism prevails across all serverless applications, which have different data dependencies. For example, while *Direct-Passing* does not impose strict scheduling constraints of receiving functions copy data simultaneously and saturate the VM’s outgoing network bandwidth. Hence, we need a hybrid, fine-grained data passing approach that optimizes data passing for every edge of an application DAG.

Our solution: We propose SONIC, a management layer for inter-lambda data exchange, which adopts a hybrid approach of the three data passing methods (*VM-Storage*, *Direct-Passing* and *Remote storage*) to optimize application performance and cost. SONIC exposes a unified API to application developers which selects the optimal data passing method for every edge in an application DAG to minimize data passing latency and cost. We show that this selection depends on parameters such as the size of input, the application’s degree of parallelism, and VM network bandwidth. SONIC adapts

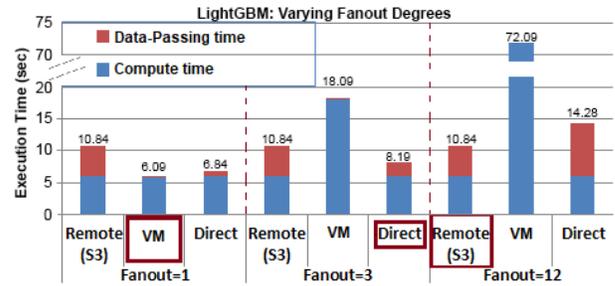


Figure 2: Execution time comparison with Remote storage, VM storage, and Direct-Passing for the LightGBM application with Fanout = 1, 3, 12. The best data passing method differs in every case.

its decision dynamically as these parameters change. Since locally optimizing data passing decisions at given stages in a DAG can be globally sub-optimal, SONIC applies a Viterbi-based algorithm to optimize latency and cost across the entire DAG. Fig. 3 shows the workflow of SONIC. SONIC abstracts its hybrid data passing selection and provides users with a simple file-based API, so users always read and write data as files to storage that appears local. SONIC is designed to be integrated with a cluster resource manager (e.g., Protean [29]) that assigns VM requests to the physical hardware and optimizes provider-centric metrics such as resource utilization and load balancing (Fig 4).

We integrate SONIC with the open-source OpenLambda [31] framework and compare performance to several baselines: AWS-Lambda, with S3 and ElastiCache-Redis (which can be taken to represent state-of-the-practice); SAND [2]; and OpenLambda with S3 and Pocket [37]. Our evaluation shows that SONIC outperforms all baselines for a variety of analytics applications. SONIC achieves between 34% and 158% higher performance/\$ (here performance is the inverse of latency) over OpenLambda+S3, between 59% and 2.3X over OpenLambda+Pocket, and between 1.9× and 5.6× over SAND, a serverless platform that leverages data locality to minimize execution time.

In summary, our contributions are as follows:

- (1) We analyze the trade-offs of three different intermediate data passing methods (Fig. 5) in serverless workflows and show that no single method prevails under all conditions in both latency and cost. This motivates the need for our hybrid and dynamic approach.
- (2) We propose SONIC, which automatically selects data passing methods between any two serverless functions in a workflow to minimize latency and cost. SONIC dynamically adapts to application changes.
- (3) We evaluate SONIC’s sensitivity to serverless-specific challenges such as the cold-start problem (set-up time for the application’s environment when it is invoked for the first time), the lack of point-to-point communication, and the provider’s lack of knowledge of lambda input data content. SONIC shows its benefit over all baselines with three common classes of serverless applications, with different input sizes and user-specified parameters.

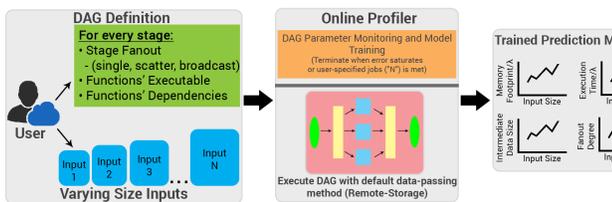


Figure 3: Workflow of SONIC: Users provide a DAG definition and input data files for profiling. SONIC’s online profiler executes the DAG and generates regression models that map the input size to the DAG’s parameters. For every new input, the online manager uses the regression models to identify the best placement for every λ and best data passing method for every pair of sending/receiving λ s in the DAG

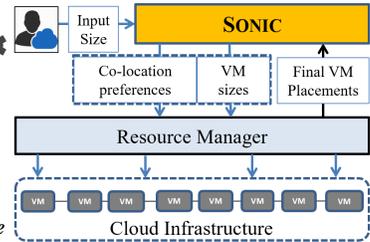


Figure 4: SONIC’s interaction with an existing Resource Manager in the system.

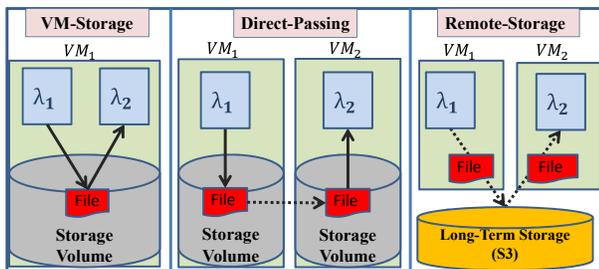


Figure 5: The three data passing options between two lambdas (λ_1 and λ_2). *VM-Storage* forces λ_2 to run on the same VM as λ_1 and avoids copying data. *Direct-Passing* stores the output file of λ_1 in the source VM’s storage, and then copies the file directly to the receiving VM’s storage. *Remote storage* uploads and downloads data through a remote storage (e.g., S3).

2 Rationale and Overview

We discuss the trade-offs of SONIC’s three data passing methods and motivate our hybrid and dynamic approach.

2.1 data passing Methods

SONIC chooses from among the three data passing options shown in Fig. 5 for *each* edge in the application DAG.

VM-Storage: This method saves the local state of the sending λ in the VM’s storage and schedules the receiving $\lambda(s)$ to execute on the same VM. This method leverages data locality to minimize latency, but imposes scheduling constraints. If the sending VM’s memory cannot fit all receiving λ s, this method forces the scheduler to run the receiving λ s serially or in batches, sacrificing parallelism in favor of data locality. This method is infeasible if the memory requirement of a single receiving λ exceeds the VM’s capacity, or when the receiving λ collects data from multiple λ s running on different VMs. Moreover, this data passing method may not be preferred by the resource manager in high load scenarios, where spreading the receiving functions over many servers is needed to avoid hot-spots and achieve better load balancing.

Direct-Passing: This data passing method saves the output of the sending λ on its VM storage and sends the λ ’s access information (IP address and File Path) to SONIC’s metadata manager. When one of the receiving λ s is scheduled to execute, the metadata manager uses the saved access information to copy the data file *directly* to the destination VM with the receiving λ . *Direct-Passing* allows higher degrees of paral-

lelism and poses no restrictions on λ placements compared to *VM-Storage*, but requires data to be sent over the network between source and destination VMs.

Remote-Storage: This data passing method involves uploading output files to a remote storage system and downloading them at destination $\lambda(s)$. This is the state-of-practice in commercial serverless platforms and has been optimized in several recent papers [12, 37, 53]. This method provides high scalability with no restrictions on λ placement. It also has the advantage of almost uniform data passing time with increasing fanout degrees as shown in Fig. 6 due to the high bandwidth of the storage layer. The disadvantage of *Remote-Storage* is having two serial data copies in the critical path — one from the source lambda to the remote storage and one from the remote storage to the destination lambda.

2.2 Dynamic Data Passing Method Selection

The optimal choice of data passing method for a job depends on the DAG’s parameters, which can vary due to dynamic conditions such as the input size, changes in network bandwidth, or changes in user-specified parameters (e.g., degree of parallelism). For example, we run the LightGBM application (application details are given in §5.3) with varying degrees of fanout and find that *VM-Storage* achieves optimal end-to-end execution time when the fanout is low (Fig. 2). However, when the maximum degree of parallelism across all stages is three, *VM-Storage* requires executing functions serially to fit on the same VM, making *Direct-Passing* superior. With a sufficiently high fanout, the sending VM faces a network bottleneck, making *Remote-Storage* the optimal data passing mechanism. Hence, no single data passing method prevails under all conditions.

SONIC prefers the *VM-Storage* method in cases where the receiving $\lambda(s)$ can be scheduled on the same VM as the sending λ while executing in parallel. However, in cases where *VM-Storage* is infeasible or sacrifices parallelism, SONIC selects between *Direct-Passing* or *Remote-Storage* methods. This selection depends on two factors: (1) VM’s network bandwidth (2) the fanout (i.e., parallelism) type and degree. To understand how these two parameters impact the selection between *Direct-Passing* and *Remote-Storage*, we show an experiment on the LightGBM application, which has a Broadcast Fanout stage (identical data from the sending λ is sent

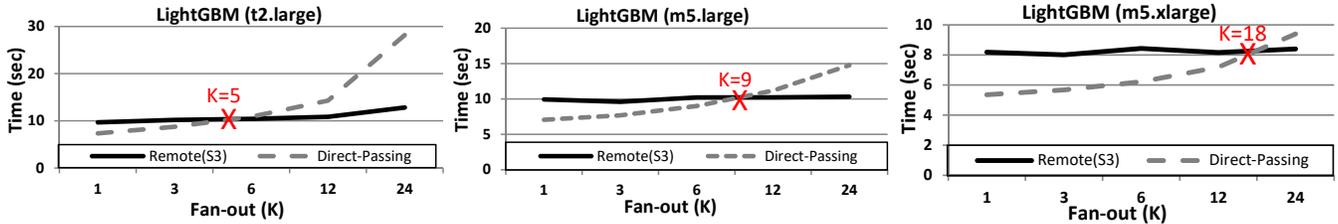


Figure 6: Comparison between Remote-Storage and Direct-Passing for LightGBM workload with varying fanout degrees. Typically, beyond a certain fanout, Remote has lower execution time than Direct-Passing. A more well-provisioned VM (m5.xlarge) will shift the crossover point to the right.

to all the receiving λ s). We vary the fanout degree (K) while using different VM types with varying network bandwidths in Fig. 6. With low degrees of parallelism (i.e., low values of K), *Direct-Passing* achieves lower latency than *Remote-Storage*. This is because copying the data directly across the two VMs is faster than copying the data twice over the network, to and from the remote storage. However, with increasing values of K , *Direct-Passing* suffers from the limited network bandwidth of the VM of the sending λ , which becomes the bottleneck as it tries to copy the intermediate data to K VMs simultaneously. In contrast, *Remote-Storage* can provide faster data passing in this case, since the sending λ saves its output file once to the remote storage and then every receiving λ downloads that file simultaneously. Thus, *Remote-Storage* provides better scalability over *Direct-Passing* in cases of large fanout degrees. The cross-over point shifts to the right as we go to more well-resourced VMs, from a network bandwidth standpoint (m5.xlarge > m5.large > t2.large). From this example, we see that selecting the best data passing method depends on the VM’s network bandwidth and DAG parameters.

We identify a number of trade-offs to be considered when performing this optimization. *First*, a trade-off between data locality and parallelism arises when the available VM’s compute and memory capacities are not sufficient to execute all receiving λ (s) in parallel. This preference of data locality over parallelism can be beneficial for lightweight functions communicating large volumes of data, while it can be harmful for compute-heavy functions with small volumes of data. *Second*, executing functions serially has the benefit of avoiding cold-start executions, which can significantly increase the execution time for functions that need to fetch many dependencies before execution [58]. *Finally*, we differentiate between two types of fanout stages: Scatter and Broadcast, depending on whether the output data is split equally among all outgoing edges (Scatter) or the same data is sent on all outgoing edges (Broadcast). With Scatter fanout, the intermediate data volume being sent is constant with the fanout degree while with Broadcast fanout, the intermediate data volume being sent increases linearly with the fanout degree. Therefore, the optimal data passing method also depends on the type of fanout and thus we cover both in our evaluation applications (video analytics and MapReduce sort for Scatter fanout and LightGBM for Broadcast fanout). In the next section, we describe the design of each component in SONIC.

3 Design

In § 3.1, we provide an overview of SONIC’s usage model. § 3.2 describes SONIC’s online profiling. § 3.3 discusses how SONIC estimates a job’s execution time for different input data sizes. § 3.4 shows how SONIC selects globally optimized data passing decisions. Finally, § 3.5 highlights further design considerations.

3.1 Usage Model

We discuss SONIC’s usage model from the perspective of application developers and the cloud provider’s resource manager.

Application DAG: As shown in Fig. 3, application developers provide SONIC with an application represented as a DAG. We assume users execute application DAGs multiple times with potentially different input data. Each such execution instance is referred to as a *job*. SONIC does not require the FaaS provider to have access to the source code for the serverless functions or hints about the resource utilization of these functions. The DAG definition from the user includes: (1) an executable for every λ ; (2) the dependencies between λ s; (3) the fanout type in every stage (i.e., single, scatter, or broadcast). Users can also provide an upper limit on the execution time or \$ budget for a single job or a batch of jobs. Notice, SONIC does not require users to specify the memory requirements for the functions in the DAG (this is a well-known pain point for users of serverless frameworks [20, 54, 57]). This is because SONIC predicts the memory footprint for every function, using our simple regression modeling (§ 3.2) and selects the right host VM size accordingly.

Data Passing Interface: SONIC abstracts the selection of data passing methods from application developers. λ functions write intermediate data to files using a standard file API (read and write), like writing to local storage. All λ s within a job share a file namespace and if an application DAG has an edge $\lambda_s \rightarrow \lambda_r$, SONIC ensures that λ_r reads from the same file path that λ_s wrote to. It also ensures that all of a λ_r ’s input files are present in its local storage before it starts execution.

Resource Manager: SONIC’s target is to minimize communication latency and cost, which are *user-centric* metrics. However, optimizing *provider-centric* metrics (e.g., load-balancing, resource utilization, and fairness) is also important. Fortunately, many resource management systems such

as Graphene [25], Protean [29], AlloX [38] and DRF [23] are designed to optimize *provider-centric* metrics efficiently, while respecting the dependencies between the functions (i.e., parent-child execution order). Hence, SONIC is designed to integrate with a system from this category (which we refer to as *Resource Manager*), as shown in Figure 4. First, the user executes the DAG with a new input. Second, SONIC uses the input size information to predict the memory footprints, execution times, and intermediate-data sizes for the DAG, which are key DAG parameters in selecting the best VM size for every function and the best data passing method for every edge in the DAG. Finally, the Resource Manager uses SONIC’s hints and decides which VMs to allocate on the available physical machines, and responds to SONIC with the final placement information, i.e., which functions go on which VMs. The Resource Manager may override SONIC’s hint to use *VM-Storage* whenever its scheduling constraint is not acceptable to the manager. Therefore, for that selection, SONIC always proposes the second-best option.

3.2 Online Profiling and Model Training

SONIC profiles jobs *online* to determine the impact of changes in input size to the following parameters: (1) each λ ’s memory footprint, (2) each λ ’s compute time, (3) intermediate data volume between any two communicating λ s, and (4) the fanout type and degree in every stage. Since the above parameters are *not* dependent on the data passing method, initially SONIC uses a default data passing policy (*Remote Storage*) while it collects DAG parameters for the first N runs of the job to train its models. N is either the number of jobs needed to reach convergence (default) or explicitly set by the user. Next, SONIC trains a set of prediction models that estimate the DAG parameter values for new inputs. SONIC develops polynomial regression models and splits the data collected into training and validation sets, then performs 5-fold cross validation to find the best model to avoid overfitting. We use polynomial regression models as they can learn from limited data points, are lightweight, and are interpretable [8].

"Online" profiling means that SONIC serves workloads while the models are being trained, which is important for practical adoption in production environments. In discussions with commercial cloud providers, we have repeatedly sensed an anathema to solutions that require offline training due to the concern that one will constantly be taking the system offline to (re-)train the models. However, if the system owner has ready access to representative input traces, she can feed SONIC with these representative traces *offline* to initialize the prediction models and reduce the online training burden.

SONIC measures the compute time for every λ under two conditions: cold execution (i.e., a new VM/container needs to be created) and hot execution (i.e., a warm VM/container with pre-loaded models/dependencies already exists). SONIC uses the difference between the two execution times in deciding

whether to queue λ s on the same VM and sacrifice parallelism (hot execute), or to execute the λ s on different VMs in parallel with the additional data passing cost and startup latency (cold execution).

3.3 Minimizing End-to-End Execution Time

We estimate the execution time τ of a stage S_i as follows:

$$\tau(S_i) = \text{DataPass}(S_{i-1}, S_i) + \text{Compute}(S_i) \quad (1)$$

When *VM-Storage* is selected, all λ s in a given stage are forced to run on the same VM. If only one λ can run at a time, the first λ incurs a cold execution time and all subsequent λ s experience hot executions. SONIC estimates τ_{VM} as follows:

$$\tau_{VM}(S_i) = 0 + \text{Cold}(\lambda_{i,1}) + \sum_{j=2}^K \text{Hot}(\lambda_{i,j}) \quad (2)$$

Here we set $\text{DataPass}(S_{i-1}, S_i)$ to zero since no additional latency is incurred by *VM-Storage* passing. For simplicity, we give here the upper bound, if all the λ s are serialized. In practice, we estimate based on batches that are serialized and all λ s within a batch can run concurrently. For *Direct-Passing* and *Remote Storage*, we take the nature of the fanout type into account. For *Direct-Passing*, with Broadcast-type fanout, the runtime is given by:

$$\tau_{\text{Direct}_B}(S_i) = \frac{F \times K}{\min(\text{BW}(VM_{i-1}), \text{BW}(VM_i))} + \text{Cold}(\lambda_{i,1}) \quad (3)$$

Where F is the intermediate data size, K the fanout, and $\text{BW}(VM_i)$, the bandwidth of the VM type hosting λ s in the i -th stage. Notice that the bandwidth is limited by the slowest of the sending and receiving VMs and that all execution times are *cold*, yet only one execution is accounted for, since they all run in parallel. For Scatter-type fanout, the equation becomes:

$$\begin{aligned} \tau_{\text{Direct}_S}(S_i) &= \frac{F/K}{\min(\text{BW}(VM_{i-1})/K, \text{BW}(VM_i))} + \text{Cold}(\lambda_{i,1}) \\ &= \frac{F}{\text{BW}(VM_{i-1})} + \text{Cold}(\lambda_{i,1}) \end{aligned} \quad (4)$$

Often cloud providers overprovision network bandwidth [14], hence we assume location of the source and destination VMs (intra- vs. inter-rack) does not affect the network bandwidth. Also, we find that the bandwidths are symmetric between VMs for all VM types; however, for remote storage, writing was faster than reading. Next, we show how we use the previous equations for our optimization.

3.4 Online VM and Data Passing Selection

A major challenge to optimize the Perf/\$ for the entire DAG is to do local selections for *each* stage to achieve the global optimum. Consider Fig. 1 for example: if we used *VM-Storage* passing between *Split_Video* and *Extract Frame*, all the extracted frames will reside in a single VM. Selecting *VM-Storage* between *Extract Frame* and *Classify Frame* will force *Classify Frame* to execute serially since that single VM does not have enough memory. However, if we select *Direct-Passing* or *Remote Storage* passing between

Split_Video and Extract Frame, every extracted frame will now reside in a separate VM. Therefore, we can use VM-Storage between Extract Frame and Classify Frame without sacrificing parallelism, lowering the DAG's execution time. Thus, greedily optimized decisions for individual stages can lead to sub-optimal global DAG performance.

To overcome the issue of sub-optimal greedy decisions, we apply the Viterbi algorithm [18] to find the globally optimized solution. SONIC uses a recursive scoring algorithm to generate all possible lambda assignments in every stage in the DAG based on the VM's compute and memory capacities, along with the λ s' predicted memory footprint. SONIC explores all possible λ -placement options under the constraints of VM resources (CPU and memory primarily) and selects the best data passing method for every pair of stages. Then, SONIC constructs a dynamic programming table with all the generated solutions. Finally, SONIC applies the Viterbi Algorithm to find the best sequence of options (i.e., the optimum Viterbi path) in the dynamic table. The selected solution is the one with the best Perf/\$ that also meets the user bounds on execution time and cost budget. This approach relies on the fact that the execution time up until stage i is equal to the execution time to stage $i - 1$ + data passing time between the two stages. This makes the Markovian assumption true (next state depends only on the current state) and makes the computation tractable.

We choose the Viterbi algorithm as it is guaranteed to find the true maximum a posteriori (MAP) solution [18, 19] unlike heuristic-based searching algorithms such as Genetic Algorithms or Simulated Annealing. The runtime complexity of the algorithm is $O(P^2 \times S)$, where P is the number of feasible λ placements on VMs for a given stage and S is the number of stages. P is upper bounded by the degree of parallelism of the stage (e.g., AWS sets a limit of 1,000) and in practice is much smaller considering many co-locations on VMs are infeasible. The runtime increases with the number of stages in the DAG, not the number of nodes or edges or fanout degree. This is desirable as the number of stages is small in practice, where a DAG of 8 stages is considered long for current serverless applications [53]. This reduction in complexity happens because SONIC applies the *same* data passing method to *all* the functions in a given stage.

3.5 Further Design Considerations

Fault tolerance. Most FaaS providers apply an automatic retry mechanism upon execution failure (e.g., AWS Lambda, Google Functions, and Azure Functions) to ensure that functions are executed *at-least-once*. For this retry mechanism to be successful, functions are required to be idempotent. To achieve idempotence, SONIC's file API assigns an ID to every intermediate file in the DAG that is used by the function that writes that file. Hence, a re-execution of this function simply overwrites the files from the previous execution. However,

as highlighted in RAMP [7] and AFT [60], idempotence in itself is not sufficient to achieve fault tolerance in serverless environments, since it does not guarantee *atomic visibility*. The problem happens when a sender λ generates only a subset of its output files, and then fails, triggering an incomplete subset of receiving λ s, resulting in a corrupted state. Therefore, SONIC applies the concept of *atomic visibility* by delaying the execution of *all* receiving λ s that belong to the same logical request until *all* their input files are successfully written to storage (either EBS in case of *Local-VM or Direct-Passing*, or S3 in case of *Remote Storage*). Although this delaying mechanism can potentially increase the E2E latency of the DAG, our evaluation shows that this additional latency is negligible: 6.3% for MapReduce, 3.3% for Video-Analytics, and 0.5% for LightGBM. All the SONIC results in the evaluation are with atomicity enabled.

DAGs with Content-Sensitive Structures. Our design assumes that the stages in the DAG are static and known, while the fanout degree in every stage can vary based on the input. This is analogous to state machines created by serverless orchestrators such as AWS Step-Functions [4]. Our video-analytics application is an example of an input-dependent fanout degree DAG and SONIC successfully predicts the fanout for new input sizes. If the structure of the DAG's stages depends on the input content, then our estimate of the DAG will be inaccurate. We evaluate the sensitivity of Sonic to prediction errors in Sec. 5.6.3.

Scalability of SONIC. For scalability, SONIC adopts a simple distributed design in which no state is shared across application jobs. When a new job arrives, SONIC's centralized component makes the decision whether to use an existing instance (if it is not overloaded) or to spin up a new instance to handle the new job. There is a central scoreboard maintained to keep track of the number of jobs being handled by each instance. The central component is lightweight in terms of its computational load and state. However, should it be necessary, this itself can be distributed through standard state machine replication (SMR) strategies [44].

4 Implementation

We implement SONIC as a data passing management layer and we use OpenLambda as our serverless platform [31]. OpenLambda is an open-source platform that relies on Linux containers for isolation and orchestration [51]. We choose OpenLambda for its flexibility—SONIC needs to control the lambda placement and needs IP addresses and administrative access to the hosting VMs. This level of control is infeasible to achieve on AWS Lambda or any other commercial offering. We implement SONIC in C# (482 LOC) and deploy it on EC2 instances, providing the same isolation guarantees as AWS Lambda across users [6]. In our setup of SONIC on OpenLambda, we use one separate container for each function (OpenLambda's design), while one VM can host multiple containers for the same application.

SONIC’s data manager consists of two parts: a centralized manager that stores IP addresses and file paths, and a distributed manager, deployed in every VM, executing the SONIC-optimized data passing method. The distributed manager also measures the actual DAG parameters during the online phase and sends them to the online manager, which updates the regression models in an incremental fashion. Moreover, since the network bandwidth can vary over time, we monitor it using Cloudwatch [13] (default of every 5 min). We use a weighted average of historic measures (as is common [56, 63]) when estimating data passing times. Thus SONIC adjusts its decisions based on bandwidth fluctuations. We use EBS storage as our storage for *VM-Storage* and *Direct-Passing* methods. EC2 EBS-optimized instances (e.g., our choice m5) have dedicated bandwidth for EBS I/O (minimizes network contention with other traffic), and can be rightsized for the predicted intermediate data volume.

5 Evaluation

5.1 Performance Metrics

Our primary performance metric is Perf/\$. Since we are minimizing end-to-end (E2E) execution time (i.e., latency), this is given by: $\frac{1}{\text{Latency}(\text{sec})} \times \frac{1}{\text{Price}(\$)}$. We also use raw latency as a secondary metric, to separate the \$-cost normalization effect. For the first metric, higher is better, and for the second, lower is better. When we refer simply to performance, we mean Perf/\$. When we refer to the secondary metric, we explicitly say (E2E) execution time or latency.

The different data passing methods considered by SONIC vary in their billing cost, which is sensitive to the selected configurations for each method. Hence, our solution should consider both latency **and** cost when selecting the best data passing method. Consequently, we use the Perf/\$ metric. We also empirically demonstrate that SONIC performs comparably to the baselines in terms of raw performance, and in many cases, outperforms the baselines (Figures 7, 8, 13, 14). Finally, there is precedence of prior work in cloud optimization using performance normalized by price [32, 40, 62]. SONIC’s *VM-Storage* and *Direct-Passing* methods add additional cost due to the extra local storage (e.g., EBS storage). However, this additional price is comparable to that of remote storage such as S3, and we include it in our evaluation.

5.2 Baselines and Methodology

We compare SONIC to the following baselines:

1. **OpenLambda + S3** [31]: This is the OpenLambda framework deployed on EC2 with S3 as its remote storage. A new VM is created to host each λ in the DAG. The smallest VM that has enough memory to execute the λ is selected.
2. **OpenLambda + Pocket** [37]: This is a variant of the OpenLambda framework with Pocket (deployed in EC2) as the remote storage. We use Pocket’s default storage tier

(DRAM) with *r5.large* instance types. The DRAM storage tier strikes the balance between performance and cost and provides the best Perf/\$ (Table 4 in [37]). Comparing the price for one DRAM node to one SSD node in Pocket, DRAM is also the cheapest tier. We vary the number of Pocket nodes for each application to reach the best Perf/\$. We include the price of the storage nodes only, excluding master and meta-data nodes. We use EC2’s per-second level pricing.

3. **SAND** [2]: This baseline leverages data locality by allocating all lambda functions on a single host with rich resources and performing data passing between chained functions through a local message bus.
4. **AWS- λ** : The commercial FaaS platform using two different remote storage systems: S3 and ElastiCache-Redis.
5. **Oracle SONIC**: This is SONIC with fully accurate estimation of DAG parameters and no data passing latency (mimicking local running of all functions). Although impractical, this serves as an upper-bound on performance for any of the three data-exchange techniques selected by SONIC.

Similar to prior OpenLambda evaluations [50], we deploy OpenLambda (and SONIC) on AWS EC2 General Purpose instances (*m5.large*, *m5.xlarge*, *m5.2xlarge*) for their balance between CPU, memory, and network bandwidth. We use the same EC2 family with SAND for fairness. *m5* instances have a network bandwidth of upto 10 Gbps, which we rely on for both *Direct-Passing* and *Remote-Storage*. All costs follow pricing by Amazon for 01/2021, N. California (Region).

5.3 Applications

We use three analytics applications popular as serverless applications and that span the variety of DAG structures.

Video Analytics: Fig. 1 shows the DAG for the Video Analytics application, which performs object classification for frames in a given video. It starts with a lambda that splits the input video into chunks of fixed length (10 sec in our case). Then, a second lambda is called for every chunk to extract a representative frame. Next, a third lambda uses a pre-trained deep learning model (MXNET [49]) to classify the extracted frame. It outputs a probability distribution across 1,000 classes over which MXNET is trained. Finally, all the classification results are written to long-term storage.

LightGBM: This application trains decision trees, combining them to form a random forest predictor using LightGBM Python library [39]). First, a λ reads the training examples and performs PCA. Second, a user-specified number of λ s train the decision trees in parallel (every λ randomly selects 90% for training, 10% for validation). A third λ collects and combines the trained models; then tests the combined model on held-out test data. Handwritten images’ databases: NIST (800K images); MNIST (60K images) used as inputs [15, 26]. **MapReduce Sort**: This application implements MapReduce sort with serverless functions. In the first stage, K parallel lambdas (i.e., mappers; K = user parameter) fetch the input

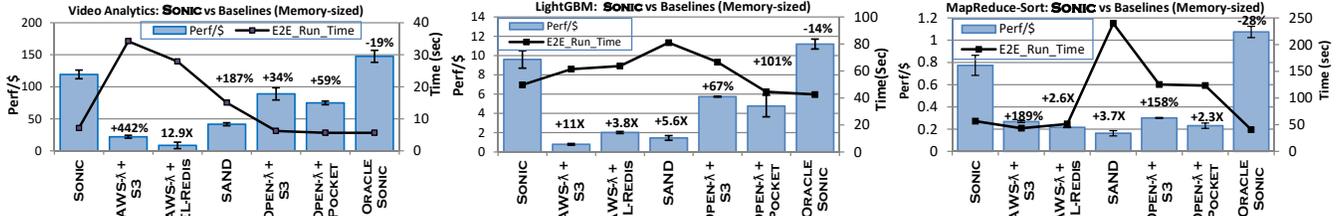


Figure 7: Performance of SONIC and the baselines for our three applications (memory-sized). Two performance metrics are shown: Perf/\$ (bars; left axis) and end-to-end execution time (lines; right axis). Relative improvements in Perf/\$ due to SONIC are at the top of each bar for the corresponding baseline.

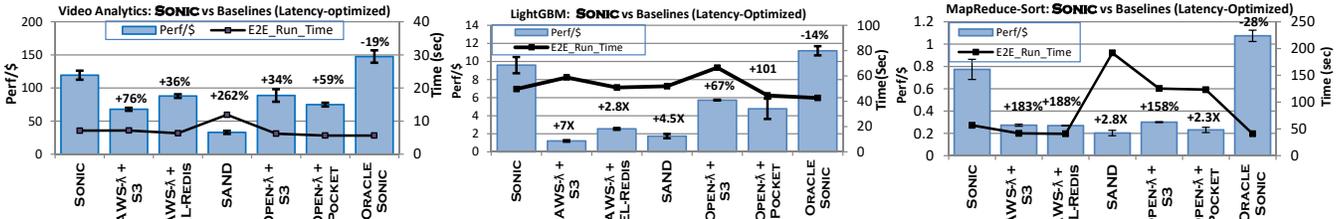


Figure 8: Performance of SONIC and the baselines for our three applications with the latency-optimized configuration.

file from a remote data store (e.g., S3) and then generate the intermediate files. Next, K parallel lambdas sort the intermediate files and write their sorted output back to storage.

5.4 End-to-End Evaluation

We compare the E2E Perf/\$ and latency of SONIC vs. other baselines for our three applications. For Video Analytics, we use a video of length 3 min and size of 15MB, which generates a fanout degree of 19 (we vary the video size in § 5.6.3). For LightGBM, we use an input size of 200MB and fanout degree of 6, generating a random forest of 6 trees (we vary the input size in § 5.5.2). For MapReduce Sort, we use an input size of 1.5GB and a fanout degree of 30. All the results of SONIC include its overheads (11ms for the DAG parameter inference phase; 120ms for the Viterbi optimization phase).

We perform online profiling and training with 35 jobs, by which we reach convergence for all applications with a low average Mean Absolute Percentage Error (MAPE) $\leq 15\%$. We show the accuracy of SONIC's predictions in Section. 5.6.1.

Memory configurations. SONIC infers memory requirements automatically from online profiling. Here, we describe how we select the memory allocation for each baseline. AWS Lambda and other serverless offerings scale compute resources relative to a user-specified memory allocation. We run the baselines under two different configurations, which we call "memory-sized" and "latency-optimized". These are respectively shown in Fig. 7 and Fig. 8. For the memory-sized configuration, we give each lambda just enough memory that it needs to execute. We determine each λ 's memory requirement by measuring the actual memory used when executing the DAG once with all λ s using the maximum memory limit (3GB for AWS- λ). For the latency-optimized configuration, we progressively increase the memory allocation as long as a reduction in latency is observed. We report the results for the run with the lowest latency. For OpenLambda, we use

SONIC's memory footprint predictor to select the cheapest VM that fits each lambda.

We draw several conclusions. First, although AWS- λ allows users to rightsize the allocated memory for their applications, it does not always lead to the best Perf/\$ or latency. For example, with Video Analytics and memory-sized configuration, SONIC achieves 442% and 12.9 \times better Perf/\$ over AWS- λ with S3 and ElastiCache-Redis respectively. However, with (memory) over-provisioning, the latency-optimized configuration improves AWS- λ performance significantly, reducing the gains of SONIC to 76% and 36% with S3 and ElastiCache-Redis respectively (similar observation is shown w.r.t. raw latency). The reason is that AWS- λ allocates all other resources (e.g., CPU capacity, network bandwidth, etc.) proportionally to the selected memory requirement [3]. Therefore, as the allocated memory is increased, the latency decreases and the cost also increases. But the latency decreases faster than the cost increases, thus Perf/\$ increases. However, beyond a certain point of over-provisioning, the latency does not decrease further, and thus the Perf/\$ begins to decrease.

For LightGBM and MapReduce Sort applications, we do not see a significant improvement in Perf/\$ for AWS- λ baselines with the latency-optimized configuration, since the memory footprint of these applications is close to the 3GB limit to begin with leaving very little room for over-provisioning. For AWS- λ , using ElastiCache-Redis as the remote storage achieves 18% lower latency than using S3. However, ElastiCache-Redis increases the cost significantly, causing a *reduction* of Perf/\$.

Compared to SAND, SONIC achieves 187% better Perf/\$ with 2 \times lower latency in the memory-sized case for Video Analytics application. The gain increases to 5.6 \times and 3.7 \times for LightGBM and MapReduce Sort applications, respectively. This is again due to the higher memory footprints of these two applications compared to Video Analytics. This reduces SAND's ability to run more λ s in parallel and forces a high

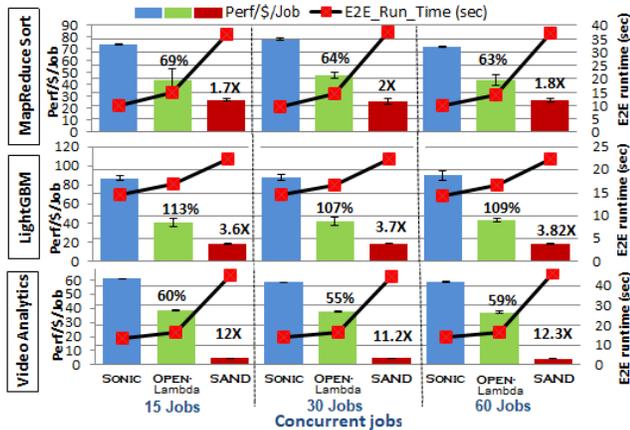


Figure 9: Scalability of SONIC, OpenLambda+S3, and SAND with equal portions of our three apps running concurrently. SONIC maintains its improvement over the entire scale, in both Perf/\$/job and raw latency.

degree of serialization. This explains the spike in latency for SAND in both applications.

Compared to OpenLambda, SONIC achieves 34% and 59% improvement in Perf/\$ with S3 and Pocket, respectively, for Video Analytics. The performance with Pocket suffers compared to vanilla OpenLambda (i.e., with S3) due to Pocket’s higher-cost storage (e.g., r5.large) without proportional benefit. Note that for OpenLambda, its performance turns out to be identical for the memory-sized and latency-optimized configurations as increasing the memory beyond SONIC’s predicted values for each function gives no latency benefit. Finally, Oracle-SONIC outperforms SONIC, as expected, but not hugely — within 19%-28% across all applications. Recall that Oracle-SONIC has perfectly accurate predictors and assumes no data passing latency.

5.5 Scalability

Here, we evaluate SONIC’s ability to scale to concurrent invocations of a mix of the applications and larger data sizes.

5.5.1 Varying Degree of Concurrency

We compare SONIC to SAND and OpenLambda+S3 serving a mixture workload of our three applications, with equal portions of invocations (jobs) per application. We use a cluster of 52 VMs of type *m5.large* with 2 compute cores per VM. We select SAND as it always prefers to keep data local, while OpenLambda+S3 always uses *Remote-Storage* data passing. OpenLambda+S3 is also the closest baseline to SONIC in terms of performance (Fig. 7 and 8). We vary the level of concurrent invocations from 15 (5 per app) to 60 (20 per app). With 60 concurrent app invocations, the cluster executes a total of 480 functions in parallel and all compute cores are fully utilized (except for SAND). We show the Perf/\$/job and E2E runtime for every app in Figure 9². We notice that the

²We normalize by the number of jobs because naturally the total cost increases with the number of jobs completed and the normalization brings out

gain of SONIC over both baselines is consistent across the different degrees of concurrency, which shows SONIC’s ability to seamlessly scale with the number of concurrent invocations. It is of course not surprising that the VM infrastructure scales up — the question was would SONIC scale up as well. This experiment answers that question in the affirmative (within the scales of the experiment), for SONIC as well as for the two baselines. This is expected from the design of SONIC where most of its components are stateless and different instances are spun up to handle more input jobs (Section 3.5). Since SAND schedules the entire job to execute on a single VM, it cannot utilize all the compute cores and hence suffers from very high latency. In contrast, OpenLambda uses *Remote-Storage* passing between functions, which allows scheduling functions on different VMs and utilizing the available compute resources. SONIC’s hybrid approach achieves both lower E2E execution time along with high resource utilization, and therefore, achieves better Perf/\$ and raw latency than all baselines.

5.5.2 Varying Intermediate Data Size

In Fig. 12, we show the impact of changing the intermediate data size on SONIC’s performance vis-à-vis two baselines. SONIC achieves lower E2E latency and higher Perf/\$ across all input sizes by predicting the corresponding DAG parameters and selecting the best co-location of lambdas and data passing methods. Second, the normalized Perf/\$ of SONIC and OpenLambda+S3 increases with higher input sizes. The reason is that the compute:data passing time ratio for this application increases with higher input sizes. This, in turn, is because the data passing time increases linearly with the input size, whereas its computation time grows faster than linear (PCA has quadratic compute complexity [65] and it dominates the total computation time). Recall that the Oracle assumes no data passing latency but it counts computation time. Accordingly, the higher the compute:data passing time ratio, the lower the gap between Oracle and the baselines (except SAND that has no data passing component).

5.6 Microbenchmarks

Here we evaluate SONIC’s online training accuracy, sensitivity to input content, prediction noise, and cold-start times.

5.6.1 Prediction Accuracy with Online Refinement

As discussed in Sec. 3.2, our solution profiles jobs to characterize the relation between input size and the data passing relevant parameters. This training phase is done online while serving production jobs, and we use remote storage data passing (SONIC’s default) until convergence is achieved. We show SONIC’s accuracy in prediction of execution parameters for new input sizes across the three applications. First, we execute the DAG of each application with jobs of varying input sizes while we measure the DAG’s execution parameters (i.e., λ ’s

the important trend that the metric is flat across the scales, implying perfect scalability.

memory footprint, λ 's compute time, data volume on every edge, and fanout degree in every stage). Next, we divide the collected data into train and test sets. We fix the test set size while we vary the number of jobs used for training to show the reduction in error w.r.t. training with more jobs for every application. In each training run, we generate the regression models for all execution parameters in the DAG. For example, PCA's runtime regression estimated equation is: $Y = 0.0024 X^2 + 12.764$, and PCA's estimated memory equation is: $Y = 16 X + 185.5$, where X is the input size in MBs .

For Video Analytics, we collect 60 YouTube videos with lengths that vary uniformly between 1 min and 1 hour and belong to 5 different categories with equal representation of each: News, Entertainment, Nature, Sports, and Cartoon. We similarly execute the LightGBM and MapReduce Sort applications with 60 jobs each. For LightGBM, we use the MNIST database of handwritten digits [15]. The data has 60,000 training images and 10,000 test images. To execute the DAG with varying input sizes, we sub-sample the training data and vary the sampling rate between 5% and 100% uniformly. For MapReduce Sort, we generate randomly shuffled data and vary the size of the data between 3M records (255MB) and 12M records (1.5GB). We also vary the number of Mappers and Reducers uniformly between 5 and 50.

Increasing the number of jobs used in training expectedly reduces the prediction error for all applications. With 35 jobs we reach convergence for all predictions with a low average Mean Absolute Percentage Error (MAPE) $\leq 15\%$ as shown in Figures 10 for Video-Analytics. This low prediction error is essential for SONIC to determine the best lambda placement information and data passing decisions for new jobs with new input sizes. Optionally, users can set a higher level of acceptable error to reduce the number of training jobs.

We notice that Video Analytics incurs the highest prediction error among the three applications. This is because our collected videos vary significantly in their bitrate, which impacts the relation between the input size and the video's length. Recall that SONIC is content-agnostic, it does not consider any information that is dependent on the content of the data when it makes its prediction. Although this negatively impacts the prediction accuracy for content-dependent applications, it allows SONIC to generalize to a wide range of applications without the need for special processing for each type of application. Furthermore, policies for public cloud providers often prohibit any visibility into the client applications. We evaluate SONIC's sensitivity to input content in § 5.6.3.

5.6.2 SONIC's Performance Improvement Root Causes

Here we highlight the root causes for SONIC's improved performance over baselines SAND and OpenLambda + S3. We show an example DAG for LightGBM application along with the data passing and lambda placement decisions made by each approach in Figure 11. We also show SONIC's selected optimum Viterbi path in the table at the bottom. We run

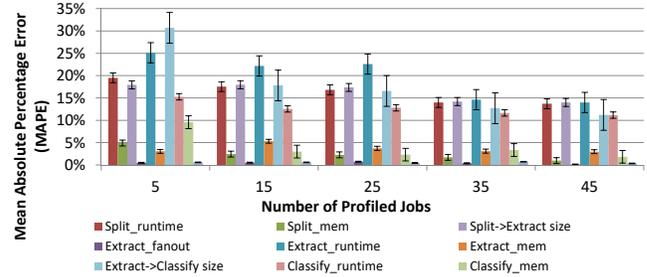


Figure 10: Error in parameter estimation for Video Analytics application. Convergence point is reached with e.g., 35 jobs. Split_mem, Extract_mem, and Classify_mem represent memory footprints for Split, Extract and Classify functions respectively. All parameters are predicted using polynomial regression models, which take the application's input size in MBs as input.

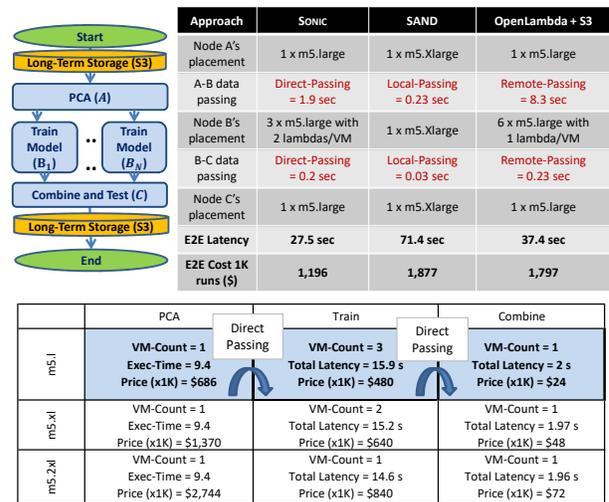


Figure 11: Example showing the benefits of SONIC over baselines. The table at the bottom shows possible λ placements for each stage in the LightGBM application and their corresponding latency and cost. We highlight the best sequence of decisions that achieves the best Perf/\$ for the entire DAG.

the LightGBM application with a fanout degree of 6 and show the E2E latency and Cost for all baselines. First, SAND leverages data locality between all stages and hence uses the same single VM of size m5.Xlarge for the entire application. This causes the fanout stage (TrainModel) to experience serialized execution as this single VM does not have enough resources to execute all invocations in parallel and hence increases the E2E latency to 71.4 sec. Compared to OpenLambda + S3, we notice that passing data between PCA and TrainModel is very slow with remote passing as it takes 8.3 sec. However, if direct passing is used (as done by SONIC), the data passing time becomes 1.9 sec only. We also notice that SONIC places each pair of lambdas in one VM using a total of 3 VMs. This makes direct passing faster as it only needs to transfer 3 copies of the transformed training data. Recall that this application has a broadcast fanout and all lambdas in TrainModel stage get the same copy of PCA's output file. In conclusion, with SONIC's optimized data passing and placement decisions, the

E2E latency is reduced by 27% over OpenLambda + S3 and by 62 % over SAND, and the Cost is reduced by 33% over OpenLambda + S3 and 36% over SAND.

5.6.3 Sensitivity to Input Content

SONIC uses only the input size information, as opposed to content awareness, to predict DAG parameters for a new job³. For example, for Video Analytics (Fig. 1) some of the parameters like the intermediate data size are sensitive to the video size (bytes), which depends on video bitrate specification. We want to examine how SONIC’s performance would be impacted on test videos different from training. In Fig. 13, we show the performance gain for three variants of SONIC, testing on a 396-sec video from the Sports category. First, we train SONIC on 60 videos from the same Sports category, which shows the best performance among the 3 variants. Second, we show SONIC’s performance with training videos from 5 different categories (60 in all, split equally), which shows an 8% performance reduction vis-à-vis the first. The third variant is trained with 60 videos from the *News* category, which has a 25% lower bitrate than the *Sports* category on average. This difference in categories causes a further performance reduction by 19% due to the error in predicting the fanout degree (40%) and intermediate data size between the `Split_Video` and `Extract_Frame` functions (21%). All three variants still show a significant gain over SAND and OpenLambda baselines. As expected, the higher the difference in critical features of the training and testing data (that can impact the DAG’s parameters), the lower is SONIC’s performance. Critical features are those that affect the compute time or the data passing volume, e.g., video bitrate. One solution to this limitation is to cluster the jobs based on the critical features and train a separate prediction model for each cluster.

5.6.4 Tolerance to Prediction Noise

We examine SONIC’s sensitivity to prediction noise. We use the MapReduce Sort application with 30 each of map and reduce functions and apply varying levels of synthetic noise to our memory footprint predictions. We show the impacts of over-predicting (i.e., the predicted memory footprint is *higher* than the actual), and under-predicting in Fig. 14. For calibration, the natural error of SONIC in prediction of memory parameters is 7%. We draw several conclusions. First, error levels of less than $\pm 20\%$ have little impact since with low levels of noise, the (categorical) decisions by SONIC for lambda-placement and data passing are unchanged. Second, under-predicting (the bars with -ve errors) has lesser impact on SONIC than over-predicting. Under-predicting causes SONIC to allocate fewer VMs (1 VM per 3 lambdas in this experiment) than without synthetic noise (1 VM per 2 lambdas). This causes the execution of only two lambdas in parallel while queuing the third lambda, increasing the job’s E2E exe-

³Although this hurts SONIC’s prediction accuracy for content-dependent DAGs, it allows generalizing without application-specific processing. Further, public cloud providers often are not allowed to look into client data.

cution time. On the other hand, over-estimation of the memory causes SONIC to allocate more VMs than what the job actually needs (1 VM per lambda). The increase in latency with under-prediction is partly compensated for by the reduction in the \$ cost, while with over-prediction, the increase in the \$ cost dominates over the reduction in latency.

5.6.5 Varying Cold-Start Overheads

In this experiment, we evaluate the effect of varying cold:hot execution times. We use a synthetic application of one stage containing 10 parallel functions and vary the function’s startup:steady state compute ratios. We compare SONIC to two static baselines, SAND and OpenLambda+S3, in Fig. 15. SAND always prefers data locality and hot execution over parallelism. We notice that this approach is beneficial for lambdas that have a gap of $5\times$ or more between cold and hot execution times. However, this solution is counter-productive when the gap between cold and hot executions is lower, unnecessarily forcing lambdas to run sequentially. The exact opposite happens with OpenLambda — it is competitive with SONIC for cold to hot execution ratios of $2\times$ or less but suffers increasingly as the ratios become higher as it always incurs cold-start costs. SONIC achieves close-to-optimal performance across the entire range of cold-to-hot execution ratios due to its ability to estimate the execution times under cold and hot executions and to select the best lambda placement and data passing approach dynamically. In practice, we find that the ratio varies in the range $[1, 3.6]$ (highest for Video Analytics’ Classify frame due to a heavy NN model); prior work has shown that the ratio can be as high as $9.6\times$ (Figure 21 in [50]). We also evaluate SONIC with varying fanout and ratios of compute time to total execution time (=compute time+data passing time). SONIC’s gain over OpenLambda is more significant at lower compute ratios as data exchange dominates, while it is more significant over SAND at higher fanouts (data locality hurts parallelism).

6 Related Work

Data-passing in serverless environments: We are not the first to identify data-passing latency as a key challenge for chained lambda execution [11, 12, 30, 31, 67]. Pocket [37] and Locus [53] implement multi-tier remote storage solutions to improve the performance and cost-efficiency of ephemeral data sharing in serverless jobs. SONIC can leverage these remote storage systems (e.g., we have evaluated SONIC with Pocket) while automatically optimizing data-passing performance with *VM-Storage* (as in SAND [2]) and *Direct-Passing* methods, which minimize data copying. Pocket also requires hints from the user about parameters of the DAG, which we infer using our modeling approach.

Prior systems have shown that serverless functions can communicate directly using NAT (network address translation) traversal techniques. ExCamera [22] uses a rendezvous server and a fleet of long-lived ephemeral workers to enable

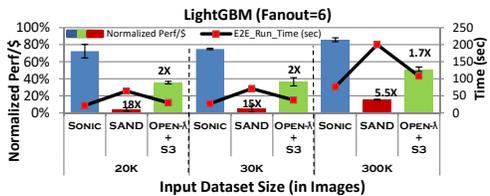


Figure 12: Normalized Perf/\$ of SONIC vs SAND and OpenLambda+S3 with varying input sizes. We fix the Fanout-degree=6 and change the number of images used in training the Random-Forest model.

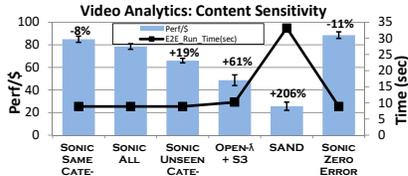


Figure 13: Effect of training SONIC on YouTube video categories similar or dissimilar to test. % over the bars represent the SONIC's (second bar from left) gain over that baseline.

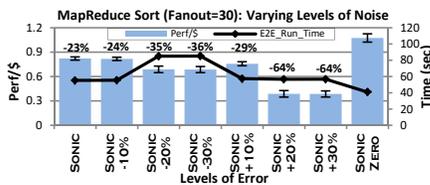


Figure 14: Impact of noise in SONIC's memory footprint predictions. Errors of less than $\pm 10\%$ have small effect and over-prediction has higher effect than under-prediction. The values over the bars are w.r.t. SONIC with zero error (rightmost).

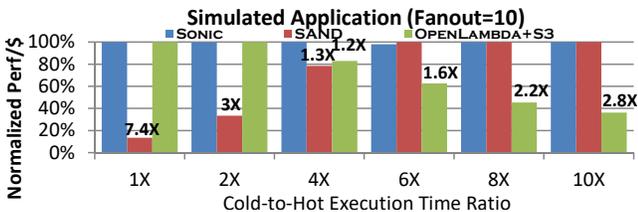


Figure 15: Effect of varying ratios of cold to hot execution times on SONIC, SAND, and OpenLambda+S3. Normalized Perf/\$ is calculated by dividing the Perf/\$ by the max across the three techniques.

direct communication. However, it needs both endpoint lambdas to be executing at the time of the data transfer, which SONIC does not. gg [21] is a framework for burst-parallel applications that supports multiple intermediate data storage engines, including direct communication between lambdas, which users can choose from. In contrast, SONIC abstracts and adaptively selects the optimal data-passing mechanism.

The need for processing state within serverless frameworks is being increasingly recognized [9, 46, 53, 59], e.g., for fine-grained state sharing or coordination among processes in ML workflows. This trend will emphasize the importance of efficient data passing among functions like SONIC provides. Automated tuning systems of clusters configurations have been proposed in [40–42]. However, these systems use black-box machine learning optimization and rely on hundreds of offline profiling runs to build accurate performance models. Cloudburst proposes using a cache on each lambda-hosting VM for fast retrieval of frequently accessed data in a remote key-value store [61], adding a modicum of statefulness to serverless workflows. SONIC does not cache data, but still exploits data locality with its lambda placement.

Efficiency of serverless executions. There is flourishing work to make serverless executions more efficient. One strategy optimizes cold-start latencies, which will influence SONIC's function placement decisions as in § 5.6.5 (e.g., SOCK [50], SEUSS [10], and Firecracker [1].) Another strategy optimizes in the isolation vs. agility spectrum (e.g., Firecracker [1], MVE [16] (supporting hugely concurrent services as in popular games), Spock [28], and Fifer [27] (hybrids of serverless and other cloud technologies for microservices). The data-passing selection in these works can benefit from SONIC. Costless [17] optimizes lambda fusion and placement, reducing the number of state transitions. This contribution is

orthogonal and beneficial to SONIC.

Cluster computing frameworks: Many distributed computing frameworks, such as Spark [66], Dryad [33], CIEL [48], and Decima [43] use DAGs of jobs to schedule tasks. With some engineering effort, SONIC can be used to support the data passing on these DAGs. However, SONIC stays close to the spirit of serverless in that it requires a minimal number of user hints or configuration options.

7 Conclusion

Optimizing the cost and performance of analytics jobs on serverless platforms requires minimizing the data passing latency between chained lambdas. The optimal data passing method depends on application-specific parameters, such as the input data size and the degree of parallelism. We presented SONIC, a system that *jointly* optimizes the inter-lambda data exchange method and lambda placement. SONIC performs online profiling to determine the relation between the application's input size and its DAG parameters. Afterward, SONIC applies an online Viterbi algorithm, to globally minimize the application's end-to-end latency, normalized by cost. SONIC achieves lower (\$ cost-normalized) latency against four competitive baselines for three popular serverless applications. Moreover, SONIC is able to adjust the best data passing method based on infrastructure changes such as network bandwidth fluctuations. In ongoing work, we are designing SONIC to handle conditional control flows in the application DAG through content-aware prediction.

8 Acknowledgement

We thank our shepherd Mike Mesnier and all the reviewers for their insightful comments. This work is supported in part by NSF grants: 1919197, 2016704, NIH Grant 1R01AI123037, Lilly Endowment (Wabash Heartland Innovation Network - WHIN), Amazon Research Awards (ARA), and Adobe Research. This material was in part based upon research supported by the U.S. Department of Energy, Office of Science, Office of Biological and Environmental Research, under contract DE-AC02-06CH11357. The funders had no role in the design or execution of the work. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

- [1] AGACHE, A., BROOKER, M., IORDACHE, A., LIGUORI, A., NEUGEBAUER, R., PIWONKA, P., AND POPA, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)* (2020), pp. 419–434.
- [2] 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), pp. 923–935.
- [3] AMAZON. Aws lambda features. <https://aws.amazon.com/lambda/features/>, 2021.
- [4] AMAZON. Aws step functions: Assemble functions into business-critical applications. <https://aws.amazon.com/step-functions/>, Last retrieved: Jan, 2021.
- [5] AO, L., IZHIKEVICH, L., VOELKER, G. M., AND PORTER, G. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing* (2018), pp. 263–274.
- [6] AWS. Aws lambda faqs. <https://aws.amazon.com/lambda/faqs/>, 2021.
- [7] BAILIS, P., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Scalable atomic visibility with ramp transactions. *ACM Transactions on Database Systems (TODS)* 41, 3 (2016), 1–45.
- [8] BANKS, D. L., AND FIENBERG, S. E. Multivariate statistics.
- [9] BARCELONA-PONS, D., SÁNCHEZ-ARTIGAS, M., PARÍS, G., SUTRA, P., AND GARCÍA-LÓPEZ, P. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference* (2019), pp. 41–54.
- [10] CADDEN, J., UNGER, T., AWAD, Y., DONG, H., KRIEGER, O., AND APPAVOO, J. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–15.
- [11] CARREIRA, J., FONSECA, P., TUMANOV, A., ZHANG, A., AND KATZ, R. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS* (2018), vol. 2018.
- [12] CARREIRA, J., FONSECA, P., TUMANOV, A., ZHANG, A., AND KATZ, R. Cirrus: a serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing* (2019), pp. 13–24.
- [13] CLOUDWATCH, A. Monitoring ec2 network utilization. <https://cloudonaut.io/monitoring-ec2-network-utilization/>, 2020.
- [14] CORTEZ, E., BONDE, A., MUZIO, A., RUSSINOVICH, M., FONTOURA, M., AND BIANCHINI, R. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 153–167.
- [15] DENG, L. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.
- [16] DONKERVLIET, J., TRIVEDI, A., AND IOSUP, A. Towards supporting millions of users in modifiable virtual environments by redesigning minecraft-like games as serverless systems. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)* (2020).
- [17] ELGAMAL, T. Costless: Optimizing cost of serverless computing through function fusion and placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)* (2018), IEEE, pp. 300–312.
- [18] FORNEY, G. D. The viterbi algorithm. *Proceedings of the IEEE* 61, 3 (1973), 268–278.
- [19] FORNEY JR, G. D. The viterbi algorithm: A personal history. *arXiv preprint cs/0504020* (2005).
- [20] FORUMS, A. How to set memory size of azure function? <https://social.msdn.microsoft.com/Forums/en-US/9a6e4728-d54a-488d-9007-5fdb80fc105e/how-to-set-memory-size-of-azure-function?forum=AzureFunctions>, 2018.
- [21] FOULADI, S., ROMERO, F., ITER, D., LI, Q., CHATTERJEE, S., KOZYRAKIS, C., ZAHARIA, M., AND WINSTEIN, K. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference* (2019), pp. 475–488.
- [22] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K. V., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)* (2017), pp. 363–376.
- [23] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant resource fairness: Fair allocation of multiple resource types. In *Nsdi* (2011), vol. 11, pp. 24–24.

- [24] GOOGLE. Cloud composer: A fully managed workflow orchestration service built on apache airflow. <https://cloud.google.com/composer>, Last retrieved: Jan, 2021.
- [25] GRANDL, R., KANDULA, S., RAO, S., AKELLA, A., AND KULKARNI, J. {GRAPHENE}: Packing and dependency-aware scheduling for data-parallel clusters. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 81–97.
- [26] GROTH, P. J. Nist special database 19 handprinted forms and characters database. *National Institute of Standards and Technology* (1995).
- [27] GUNASEKARAN, J. R., THINAKARAN, P., CHIDAMBARAM, N., KANDEMIR, M. T., AND DAS, C. R. Fifer: Tackling underutilization in the serverless era. *arXiv preprint arXiv:2008.12819* (2020).
- [28] GUNASEKARAN, J. R., THINAKARAN, P., KANDEMIR, M. T., URGANONKAR, B., KESIDIS, G., AND DAS, C. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)* (2019), IEEE, pp. 199–208.
- [29] HADARY, O., MARSHALL, L., MENACHE, I., PAN, A., GREEFF, E. E., DION, D., DORMINEY, S., JOSHI, S., CHEN, Y., RUSSINOVICH, M., ET AL. Protean:{VM} allocation service at scale. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)* (2020), pp. 845–861.
- [30] HELLERSTEIN, J. M., FALEIRO, J., GONZALEZ, J. E., SCHLEIER-SMITH, J., SREEKANTI, V., TUMANOV, A., AND WU, C. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651* (2018).
- [31] HENDRICKSON, S., STURDEVANT, S., HARTE, T., VENKATARAMANI, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Serverless computation with openlambda. In *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (2016).
- [32] HSU, C.-J., NAIR, V., MENZIES, T., AND FREEH, V. W. Scout: An experienced guide to find the best cloud configuration. *arXiv preprint arXiv:1803.01296* (2018).
- [33] ISARD, M., BUDI, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 Eurosys Conference* (March 2007), Association for Computing Machinery, Inc.
- [34] JONAS, E., SCHLEIER-SMITH, J., SREEKANTI, V., TSAI, C.-C., KHANDELWAL, A., PU, Q., SHANKAR, V., CARREIRA, J., KRAUTH, K., YADWADKAR, N., ET AL. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).
- [35] KIM, Y., AND LIN, J. Serverless data analytics with flint. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)* (2018), pp. 451–455.
- [36] KLIMOVIC, A., WANG, Y., KOZYRAKIS, C., STUEDI, P., PFEFFERLE, J., AND TRIVEDI, A. Understanding ephemeral storage for serverless analytics. In *2018 USENIX Annual Technical Conference (USENIXATC 18)* (2018), pp. 789–794.
- [37] KLIMOVIC, A., WANG, Y., STUEDI, P., TRIVEDI, A., PFEFFERLE, J., AND KOZYRAKIS, C. Pocket: Elastic ephemeral storage for serverless analytics. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 427–444.
- [38] LE, T. N., SUN, X., CHOWDHURY, M., AND LIU, Z. Allox: compute allocation in hybrid clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–16.
- [39] LIGHTGBM. Lightgbm’s documentation! <https://lightgbm.readthedocs.io/en/latest/index.html>, 2021.
- [40] MAHGOUB, A., MEDOFF, A. M., KUMAR, R., MITRA, S., KLIMOVIC, A., CHATERJI, S., AND BAGCHI, S. {OPTIMUSCLOUD}: Heterogeneous configuration optimization for distributed databases in the cloud. In *USENIX Annual Technical Conference (USENIX ATC)* (2020), pp. 189–203.
- [41] MAHGOUB, A., WOOD, P., GANESH, S., MITRA, S., GERLACH, W., HARRISON, T., MEYER, F., GRAMA, A., BAGCHI, S., AND CHATERJI, S. Rafiki: a middleware for parameter tuning of nosql datastores for dynamic metagenomics workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (2017), pp. 28–40.
- [42] MAHGOUB, A., WOOD, P., MEDOFF, A., MITRA, S., MEYER, F., CHATERJI, S., AND BAGCHI, S. {SOPHIA}: Online reconfiguration of clustered nosql databases for time-varying workloads. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)* (2019), pp. 223–240.
- [43] MAO, H., SCHWARZKOPF, M., VENKATAKRISHNAN, S. B., MENG, Z., AND ALIZADEH, M. Learning scheduling algorithms for data processing clusters. In

Proceedings of the ACM SIGCOMM (2019), ACM, pp. 270–288.

- [44] MARANDI, P. J., PRIMI, M., AND PEDONE, F. High performance state-machine replication. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)* (2011), IEEE, pp. 454–465.
- [45] MICROSOFT. Azure durable functions overview. <https://docs.microsoft.com/en-us/azure/azure-functions/durable/>, Last retrieved: Jan, 2021.
- [46] MORITZ, P., NISHIHARA, R., WANG, S., TUMANOV, A., LIAW, R., LIANG, E., ELIBOL, M., YANG, Z., PAUL, W., JORDAN, M. I., ET AL. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 561–577.
- [47] MÜLLER, I., MARROQUÍN, R., AND ALONSO, G. Lambda: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 115–130.
- [48] MURRAY, D. G., SCHWARZKOPF, M., SOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. Ciel: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (2011), NSDI'11, p. 113–126.
- [49] MXNET. Using pre-trained deep learning models in mxnet. https://mxnet.apache.org/api/python/docs/tutorials/packages/gluon/image/pretrained_models.html, 2021.
- [50] OAKES, E., YANG, L., ZHOU, D., HOUCK, K., HARTE, 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)* (2018), pp. 57–70.
- [51] OPENLAMBDA. An open source serverless computing platform. <https://github.com/open-lambda/open-lambda>, 2021.
- [52] PERRON, M., CASTRO FERNANDEZ, R., DEWITT, D., AND MADDEN, S. Starling: A scalable query engine on cloud functions. In *SIGMOD* (2020).
- [53] PU, Q., VENKATARAMAN, S., AND STOICA, I. Shuffling, fast and slow: scalable analytics on serverless infrastructure. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 193–206.
- [54] RAUPACH, B. Choosing the right amount of memory for your aws lambda function. <https://medium.com/@raupach/choosing-the-right-amount-of-memory-for-your-aws-lambda-function-99615ddf75dd>, 2018.
- [55] RAUSCH, T., HUMMER, W., MUTHUSAMY, V., RASHED, A., AND DUSTDAR, S. Towards a serverless platform for edge {AI}. In *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)* (2019).
- [56] RIBEIRO, V. J., RIEDI, R. H., BARANIUK, R. G., NAVRATIL, J., AND COTTRELL, L. pathchirp: Efficient available bandwidth estimation for network paths. In *Passive and active measurement workshop* (2003).
- [57] RIBENZAFT, R. How to make aws lambda faster: Memory performance. <https://epsagon.com/observability/how-to-make-aws-lambda-faster-memory-performance/>, 2018.
- [58] 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 20)* (July 2020), USENIX Association, pp. 205–218.
- [59] SHANKAR, V., KRAUTH, K., PU, Q., JONAS, E., VENKATARAMAN, S., STOICA, I., RECHT, B., AND RAGAN-KELLEY, J. Numpywren: Serverless linear algebra. In *ACM Symposium on Cloud Computing (SoCC)* (2020), pp. 1–14.
- [60] SREEKANTI, V., WU, C., CHHATRAPATI, S., GONZALEZ, J. E., HELLERSTEIN, J. M., AND FALEIRO, J. M. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–15.
- [61] SREEKANTI, V., WU, C., LIN, X. C., SCHLEIER-SMITH, J., FALEIRO, J. M., GONZALEZ, J. E., HELLERSTEIN, J. M., AND TUMANOV, A. Cloudburst: Stateful functions-as-a-service. <https://arxiv.org/pdf/2001.04592.pdf>, 2020.
- [62] TOOTAGHAJ, D. Z., FARHAT, F., ARJOMAND, M., FARABOSCHI, P., KANDEMIR, M. T., SIVASUBRAMANIAM, A., AND DAS, C. R. Evaluating the combined impact of node architecture and cloud workload characteristics on network traffic and performance/cost. In *2015 IEEE International Symposium on Workload Characterization* (2015), IEEE, pp. 203–212.

- [63] WANG, H., LEE, K. S., LI, E., LIM, C. L., TANG, A., AND WEATHERSPOON, H. Timing is everything: Accurate, minimum overhead, available bandwidth estimation in high-speed wired networks. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (2014), pp. 407–420.
- [64] XU, R., ZHANG, C.-L., WANG, P., LEE, J., MITRA, S., CHATERJI, S., LI, Y., AND BAGCHI, S. Approx-det: content and contention-aware approximate object detection for mobiles. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems* (2020), pp. 449–462.
- [65] YI, X., PARK, D., CHEN, Y., AND CARAMANIS, C. Fast algorithms for robust pca via gradient descent. In *Advances in neural information processing systems* (2016), pp. 4152–4160.
- [66] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., STOICA, I., ET AL. Spark: Cluster computing with working sets. *HotCloud 10*, 10-10 (2010).
- [67] ZHANG, T., XIE, D., LI, F., AND STUTSMAN, R. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing* (2019), pp. 1–12.