ISBN 978-1-939133-38-0

Proceedings of the 22nd USENIX Conference on File and Storage Technologies (FAST '24) *Santa Clara, CA, USA* *February 27–29, 2024*

conference
proceedings

22nd USENIX Conference on File and Storage Technologies (FAST '24)

Santa Clara, CA, USA
February 27–29, 2024

Sponsored by

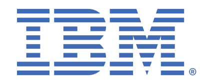
In cooperation with ACM SIGOPS

FAST '24 Sponsors

Gold Sponsors

Silver Sponsor

Bronze Sponsors



Open Access Sponsor

USENIX Supporters

USENIX Patrons
Futurewei • Google • Meta

USENIX Benefactor
Bloomberg

USENIX Partner
Thinkst Canary

Open Access Supporter
Google

Open Access Publishing Partner
PeerJ

USENIX Association

Proceedings of the 22nd USENIX Conference on File and Storage Technologies (FAST '24)

February 27–29, 2024
Santa Clara, CA, USA

© 2024 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 978-1-939133-38-0

Conference Organizers

Program Co-Chairs
Xiaosong Ma, *Qatar Computing Research Institute, Hamad Bin Khalifa University*
Youjip Won, *Korea Advanced Institute of Science and Technology (KAIST)*

Program Committee
George Amvrosiadis, *Carnegie Mellon University*
Ali Anwar, *University of Minnesota*
Oana Balmau, *McGill University*
John Bent, *Seagate*
Janki Bhimani, *Florida International University*
Angelos Bilas, *University of Crete and FORTH*
Ali R. Butt, *Virginia Tech*
Andromachi Chatzieleftheriou, *Microsoft Research*
Young-ri Choi, *UNIST (Ulsan National Institute of Science and Technology)*
Angela Demke Brown, *University of Toronto*
Peter Desnoyers, *Northeastern University*
Aishwarya Ganesan, *University of Illinois at Urbana–Champaign and VMware Research*
Ashvin Goel, *University of Toronto*
Haryadi Gunawi, *University of Chicago*
Dean Hildebrand, *Google*
Yu Hua, *Huazhong University of Science and Technology*
Jian Huang, *University of Illinois at Urbana–Champaign*
Jooyoung Hwang, *Samsung Electronics*
Jinkyu Jeong, *Yonsei University*
Sudarsun Kannan, *Rutgers University*
Sanidhya Kashyap, *EPFL*
Youngjin Kwon, *Korea Advanced Institute of Science and Technology (KAIST)*
Patrick P. C. Lee, *The Chinese University of Hong Kong (CUHK)*
Sungjin Lee, *Daegu Gyeongbuk Institute of Science and Technology (DGIST)*
Cheng Li, *University of Science and Technology of China*
Youyou Lu, *Tsinghua University*
Peter Macko, *MongoDB*
Changwoo Min, *Igalia*
Beomseok Nam, *Sungkyunkwan University*
Sam H. Noh, *Virginia Tech*
Raju Rangaswami, *Florida International University*
Jiri Schindler, *IonQ*
Phil Shilane, *Dell Technologies*
Keith A. Smith, *MongoDB*
Vasily Tarasov, *IBM Research*
Eno Thereska, *Alcion, Inc.*
Carl Waldspurger, *Carl Waldspurger Consulting*
Wen Xia, *Harbin Institute of Technology*
Gala Yadgar, *Technion—Israel Institute of Technology*
Ming-Chang Yang, *The Chinese University of Hong Kong (CUHK)*

Work-in-Progress/Posters Co-Chairs
Ali R. Butt, *Virginia Tech*
Young-ri Choi, *UNIST (Ulsan National Institute of Science and Technology)*

Test of Time Awards Committee
Geoff Kuenning, *Harvey Mudd College*
Raju Rangaswami, *Florida International University*

Mentoring Co-Chairs
Aishwarya Ganesan, *University of Illinois at Urbana–Champaign and VMware Research*
Dean Hildebrand, *Google*

Artifact Evaluation Committee Co-Chairs
Haryadi Gunawi, *The University of Chicago*
Huaicheng Li, *Virginia Tech*

Artifact Evaluation Committee
Saheed Olayemi Bolarinwa, *The Leibniz Supercomputing Centre*
Chunfeng Du, *Xiamen University*
Jian Gao, *Tsinghua University*
Andrzej Jackowski, *University of Warsaw*
Chenhao Jiang, *University of Toronto*
Ziyang Jiao, *Syracuse University*
R. Madhava Krishnan, *Oracle*
Jiamin Li, *City University of Hong Kong*
Xiaolu Li, *Huazhong University of Science and Technology*
Congyu Liu, *Purdue University*
Ruiming Lu, *Shanghai Jiao Tong University*
Teng Ma, *Alibaba Group*
Weiwu Pang, *University of Southern California*
Marcus Paradies, *Technische Universität Ilmenau*
Bo Peng, *Shanghai Jiao Tong University*
Benjamin Reidys, *University of Illinois at Urbana–Champaign*
Reza Salkhordeh, *Johannes Gutenberg University*
Tomoya Suzuki, *University of California, San Diego*
Dingwen Tao, *Indiana University*
Lingfeng Xiang, *The University of Texas at Arlington*
Danning Xie, *Purdue University*
Minhui Xie, *Tsinghua University*
Chenhao Ye, *University of Wisconsin—Madison*
Liangcheng Yu, *University of Pennsylvania*
Anlan Zhang, *University of Southern California*
Jianshun Zhang, *Huazhong University of Science and Technology*
Xuechen Zhang, *Washington State University Vancouver*
Mai Zheng, *Iowa State University*
Shawn Zhong, *University of Wisconsin—Madison*
Yuqing Zhu, *Tsinghua University*
Xiangyu Zou, *Harbin Institute of Technology Shenzhen*

Steering Committee
Marcos K. Aguilera, *VMware Research*
Ashvin Goel, *University of Toronto*
Casey Henderson-Ross, *USENIX Association*
Dean Hildebrand, *Google*
Dalit Naor, *The Academic College of Tel Aviv–Yaffo*
Sam H. Noh, *Virginia Tech*
Don Porter, *The University of North Carolina at Chapel Hill*
Raju Rangaswami, *Florida International University*
Carl Waldspurger, *Carl Waldspurger Consulting*
Gala Yadgar, *Technion—Israel Institute of Technology*

External Reviewers

Luis Ceze
David H.C. Du
Jeongseok Ha
Sang-Woo Jun
Joo-Young Kim
Jaejin Lee
Yin Yang

Message from the FAST ’24 Program Co-Chairs

Welcome to the 22nd USENIX Conference on File and Storage Technologies (FAST ’24). As the second post-COVID FAST conference, this year’s event continues the tradition of bringing together researchers and practitioners from both industry and academia for a program of innovative and rigorous storage-related research.

FAST ’24 has remained highly selective. We received 123 submissions from authors in academia, industry, government labs, and the open-source communities. Of these, we accepted 22 papers for an acceptance rate of 18%.

As usual, we have employed a two-round online review process. The first round had three reviewers assigned to each paper, followed by intensive online discussion. Forty-five papers were advanced to the second round, with an early rejection notification sent to the rest of the papers in early November 2023. The second round solicited at least two more reviews for each remaining paper and was again followed by active discussions that led to a summary request by the discussion lead listing specific items that the authors needed to respond to. This is the fourth year that FAST has allowed author rebuttal, and all 45 papers participated. More online discussions happened after the three-day rebuttal period, resulting in 9 papers being pre-accepted and 12 pre-rejected. The rest of the papers were discussed in a two-day online PC meeting in December 2023, with PC members joining virtually from global locations across 10 different time zones.

We used HotCRP to manage all the stages of the review process, from submission to author notification. A total of 464 reviews and 1742 comments were submitted on the FAST ’24 submission site. All accepted papers were assigned a shepherd from the PC, who worked with the authors to address comments from the reviews and provided editorial advice and feedback on the final manuscripts.

The review process produced a program covering a wide range of topics, including cloud and remote storage, caching, key-value stores, persistent memory and SSD systems, storage coding, learned storage systems, and new file system designs. We continued to accommodate a special category of deployed-systems papers, which share experiences with the practical design, implementation, analysis, or deployment of large-scale operational systems. We received five submissions in this category and accepted two. The program also includes posters and work-in-progress sessions.

FAST ’24 marks the first time in the conference’s history to adopt an optional artifact evaluation (AE) process, in which most accepted papers participated. After extensive evaluation, 17 papers were awarded AE badges, with about half of them receiving all three badges (“available”, “functional”, and “reproduced”).

In addition, this year’s conference maintained the FAST mentorship program designed to enhance the conference experience for student attendees. The program offers them the chance to connect with and gain valuable career insights from seasoned community members, as well as to receive constructive feedback on their research.

We are utterly thankful to the many people who contributed to this conference. First and foremost, we are grateful to all the authors who submitted their work to FAST ’24, as well as our conference attendees and future readers of the published papers. We extend our thanks to the entire USENIX staff, who have provided outstanding support throughout the planning and organization of this conference with the highest degree of professionalism and friendliness. Most importantly, their behind-the-scenes work and meticulous care of details make this conference happen. We are also grateful to KAIST students Dohyun Kim and Juwon Kim for spending many hours supporting us in configuring, testing, and managing systems used in the review process and the online PC meeting.

We would like to thank the Poster and Work-in-Progress Chairs, Ali Butt and Young-ri Choi, for managing the submission, review, and coordination of these sessions. We thank Haryadi Gunawi and Huaicheng Li for proposing and chairing the Artifact Evaluation process. We thank Aishwarya Ganesan and Dean Hildebrand for chairing the Mentorship program. Our thanks also go to the members of the FAST Steering Committee, and especially the recent FAST chairs to whom we reached out and who provided invaluable advice and feedback. We appreciate the support and suggestions from Keith Smith and Dean Hildebrand in organizing the panel session. We especially wish to acknowledge our Steering Committee Liaison, Gala Yadgar, for her continuous guidance on delicate issues, attention to things we missed, and encouragement on many occasions over the past year.

Finally, we wish to thank our Program Committee members for their many hours of hard work reviewing, discussing, and shepherding the submissions. The reviewers’ evaluations, as well as their thorough and conscientious deliberations at the PC meeting, contributed significantly to the quality of our decisions. Similarly, the paper shepherds’ efforts led to significant improvements in the final quality of the program. We look forward to an exciting and enjoyable conference!

Xiaosong Ma, *Qatar Computing Research Institute, Hamad Bin Khalifa University*
Youjip Won, *Korea Advanced Institute of Science and Technology (KAIST)*
FAST ’24 Program Co-Chairs

22nd USENIX Conference on File and Storage Technologies (FAST ’24)
February 27–29, 2024
Santa Clara, CA, USA

Tuesday, February 27

Distributed Storage

TeRM: Extending RDMA-Attached Memory with SSD 1
Zhe Yang, Qing Wang, Xiaojian Liao, and Youyou Lu, *Tsinghua University;* Keji Huang, *Huawei Technologies Co., Ltd;* Jiwu Shu, *Tsinghua University*

Combining Buffered I/O and Direct I/O in Distributed File Systems 17
Yingjin Qian, *Data Direct Networks;* Marc-André Vef, *Johannes Gutenberg University Mainz;* Patrick Farrell and Andreas Dilger, *Whamcloud Inc.;* Xi Li and Shuichi Ihara, *Data Direct Networks;* Yinjin Fu, *Sun Yat-Sen University;* Wei Xue, *Tsinghua University and Qinghai University;* André Brinkmann, *Johannes Gutenberg University Mainz*

OmniCache: Collaborative Caching for Near-storage Accelerators 35
Jian Zhang and Yujie Ren, *Rutgers University;* Marie Nguyen, *Samsung;* Changwoo Min, *Igalia;* Sudarsun Kannan, *Rutgers University*

Caching

Symbiosis: The Art of Application and Kernel Cache Cooperation 51
Yifan Dai, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau, *University of Wisconsin—Madison*

Optimizing File Systems on Heterogeneous Memory by Integrating DRAM Cache with Virtual Memory Management 71
Yubo Liu, Yuxin Ren, Mingrui Liu, Hongbo Li, Hanjun Guo, Xie Miao, and Xinwei Hu, *Huawei Technologies Co., Ltd.;* Haibo Chen, *Huawei Technologies Co., Ltd. and Shanghai Jiao Tong University*

Kosmo: Efficient Online Miss Ratio Curve Generation for Eviction Policy Evaluation 89
Kia Shakiba, Sari Sultan, and Michael Stumm, *University of Toronto*

File Systems

I/O Passthru: Upstreaming a flexible and efficient I/O Path in Linux 107
Kanchan Joshi, Anuj Gupta, Javier González, Ankit Kumar, Krishna Kanth Reddy, Arun George, and Simon Lund, *Samsung Semiconductor;* Jens Axboe, *Meta Platforms Inc.*

Metis: File System Model Checking via Versatile Input and State Exploration 123
Yifei Liu and Manish Adkar, *Stony Brook University;* Gerard Holzmann, *Nimble Research;* Geoff Kuenning, *Harvey Mudd College;* Pei Liu, Scott A. Smolka, Wei Su, and Erez Zadok, *Stony Brook University*

RFUSE: Modernizing Userspace Filesystem Framework through Scalable Kernel-Userspace Communication 141
Kyu-Jin Cho, Jaewon Choi, Hyungjoon Kwon, and Jin-Soo Kim, *Seoul National University*

Wednesday, February 28

Flash Storage

The Design and Implementation of a Capacity-Variant Storage System 159
Ziyang Jiao and Xiangqun Zhang, *Syracuse University;* Hojin Shin and Jongmoo Choi, *Dankook University;* Bryan S. Kim, *Syracuse University*

I/O in a Flash: Evolution of ONTAP to Low-Latency SSDs 177
Matthew Curtis-Maury, Ram Kesavan, Bharadwaj V R, Nikhil Mattankot, Vania Fang, Yash Trivedi, Kesari Mishra, and Qin Li, *NetApp, Inc*

We Ain’t Afraid of No File Fragmentation: Causes and Prevention of Its Performance Impact on Modern Flash SSDs 193
Yuhun Jun, *Sungkyunkwan University and Samsung Electronics Co., Ltd.;* Shinhyun Park, *Sungkyunkwan University;* Jeong-Uk Kang, *Samsung Electronics Co., Ltd.;* Sang-Hoon Kim, *Ajou University;* Euiseong Seo, *Sungkyunkwan University*

Key-Value Systems

In-Memory Key-Value Store Live Migration with NetMigrate **209**
Zeying Zhu, *University of Maryland;* Yibo Zhao, *Boston University;* Zaoxing Liu, *University of Maryland*

Ionia: High-Performance Replication for Modern Disk-based KV Stores **225**
Yi Xu, *University of California, Berkeley;* Henry Zhu, *University of Illinois Urbana-Champaign;* Prashant Pandey, *University of Utah;* Alex Conway, *Cornell Tech and VMware Research;* Rob Johnson, *VMware Research;* Aishwarya Ganesan and Ramnatthan Alagappan, *University of Illinois Urbana-Champaign and VMware Research*

Physical vs. Logical Indexing with IDEA: Inverted Deduplication-Aware Index **243**
Asaf Levi, *Technion - Israel Institute of Technology;* Philip Shilane, *Dell Technologies;* Sarai Sheinvald, *Braude College of Engineering;* Gala Yadgar, *Technion - Israel Institute of Technology*

MiDAS: Minimizing Write Amplification in Log-Structured Systems through Adaptive Group Number and Size Configuration **259**
Seonggyun Oh, Jeeyun Kim, and Soyoung Han, *DGIST;* Jaeho Kim, *Gyeongsang National University;* Sungjin Lee, *DGIST;* Sam H. Noh, *Virginia Tech*

Thursday, February 29

Cloud Storage

What's the Story in EBS Glory: Evolutions and Lessons in Building Cloud Block Store **277**
Weidong Zhang, Erci Xu, Qiuping Wang, Xiaolu Zhang, Yuesheng Gu, Zhenwei Lu, Tao Ouyang, Guanqun Dai, Wenwen Peng, Zhe Xu, Shuo Zhang, Dong Wu, Yilei Peng, Tianyun Wang, Haoran Zhang, Jiasheng Wang, Wenyuan Yan, Yuanyuan Dong, Wenhui Yao, Zhongjie Wu, Lingjun Zhu, Chao Shi, Yinhu Wang, Rong Liu, Junping Wu, Jiaji Zhu, and Jiesheng Wu, *Alibaba Group*

ELECT: Enabling Erasure Coding Tiering for LSM-tree-based Storage **293**
Yanjing Ren and Yuanming Ren, *The Chinese University of Hong Kong;* Xiaolu Li and Yuchong Hu, *Huazhong University of Science and Technology;* Jingwei Li, *University of Electronic Science and Technology of China;* Patrick P. C. Lee, *The Chinese University of Hong Kong*

MinFlow: High-performance and Cost-efficient Data Passing for I/O-intensive Stateful Serverless Analytics **311**
Tao Li, Yongkun Li, and Wenzhe Zhu, *University of Science and Technology of China;* Yinlong Xu, *Anhui Province Key Laboratory of High Performance Computing, University of Science and Technology of China,* John C. S. Lui, *The Chinese University of Hong Kong*

AI and Storage

COLE: A Column-based Learned Storage for Blockchain Systems **329**
Ce Zhang and Cheng Xu, *Hong Kong Baptist University;* Haibo Hu, *Hong Kong Polytechnic University;* Jianliang Xu, *Hong Kong Baptist University*

Baleen: ML Admission & Prefetching for Flash Caches **347**
Daniel Lin-Kit Wong, *Carnegie Mellon University;* Hao Wu, *Meta;* Carson Molder, *UT Austin;* Sathya Gunasekar, Jimmy Lu, Snehal Khandkar, and Abhinav Sharma, *Meta;* Daniel S. Berger, *Microsoft and University of Washington;* Nathan Beckmann and Gregory R. Ganger, *Carnegie Mellon University*

Seraph: Towards Scalable and Efficient Fully-external Graph Computation via On-demand Processing **373**
Tsun-Yu Yang, Yizou Chen, Yuhong Liang, and Ming-Chang Yang, *The Chinese University of Hong Kong*

TeRM: Extending RDMA-Attached Memory with SSD

Zhe Yang†, Qing Wang†, Xiaojian Liao†, Youyou Lu†, Keji Huang‡, and Jiwu Shu†*

†Tsinghua University
‡Huawei Technologies Co., Ltd

Abstract

RDMA-based in-memory storage systems offer high performance but are restricted by the capacity of physical memory. In this paper, we propose TeRM to extend RDMA-attached memory with SSD. TeRM achieves fast remote access on the SSD-extended memory by eliminating page faults of RDMA NIC and CPU from the critical path. We also introduce a set of techniques to reduce the consumption of CPU and network resources. Evaluation shows that TeRM performs close to the performance of the ideal upper bound where all pages are pinned in the physical memory. Compared with existing approaches TeRM significantly improves the performance of unmodified RDMA-based storage systems, including a file system and a key-value system.

1 Introduction

RDMA networks are catalyzing innovative designs for a wide range of in-memory storage systems, including file systems [12, 25, 41], key-value stores [26, 27, 37], and transactional databases [14, 15, 34, 38]. Unlike traditional TCP/IP networks, RDMA can expose server-side memory regions, i.e., *RDMA-attached memory*, to clients in the form of virtual addresses. Clients can directly access data in these regions via one-sided requests. The execution of one-sided requests at the server side bypasses the CPU: the RDMA NIC (RNIC) performs virtual-to-physical address translation using RNIC page table, and then DMAs data to physical memory. In this way, RDMA provides low latency and high CPU efficiency.

However, memory is an expensive and limited resource in datacenters [23, 39]. To improve cost-efficiency and accommodate larger-than-memory data sets for RDMA-based systems, it is desirable to exploit SSD to extend the space of RDMA-attached memory by performing *demand paging* with the physical memory and the SSD. A hardware mechanism called ODP (On-Demand Paging) MR (memory region) [22] is proposed to support it. When handling an RDMA request with the ODP MR, the RNIC will trigger a page fault interrupt for SSD-resident data, then the CPU promotes data from SSD to memory and updates the RNIC page table.

*Jiwu Shu is the corresponding author (shujw@tsinghua.edu.cn).

Unfortunately, our experiments demonstrate that ODP MR is not the silver bullet to extend RDMA-attached memory with SSD. As an RDMA READ consumes only 3.66μs on an in-memory page, the latency grows to 570.74μs on an SSD-resident page. The root cause is that the RNIC hardware has limited compute and memory resources [22], so it can only handle exceptions of RNIC page faults in a simple but *inefficient* manner (e.g., discard the received data and notify the client-side RNIC to retransmit it).

Motivated by the analysis above, we propose TeRM, an efficient approach to extending RDMA-attached memory with SSD. The key idea is to onload exception handling (i.e., RNIC page fault) from hardware to software. For all the SSD-resident pages, TeRM makes the RNIC page table point to a reserved physical page containing a predefined magic pattern. In this way, the RNIC page fault is eliminated. For a read request, the client first fetches data through an RDMA READ and identifies whether the page is on the SSD. Then, the client resorts to RPCs to retrieve an SSD-resident page from the server, but does not require any additional operation for memory-resident pages, ensuring fast remote accesses in common cases. Meanwhile, we introduce a set of techniques to reduce the network traffic.

The TeRM-induced RPCs will access SSD-extended virtual memory. To eliminate the heavy CPU page fault [11, 30, 42] from the critical path of RPC execution, we propose tiering IO. The key idea is to access the SSD-extended virtual memory via file IO interfaces instead of memory `load`/`store` interfaces. It reads/writes the SSD-extended virtual memory via buffer IO when the data is cached in the physical memory, and otherwise via direct IO that bypasses the page cache.

With the design techniques above, TeRM eschews both RNIC and CPU page faults from the critical path. However, it freezes data placement on the server, unfortunately. If a hotspot is on the SSD, it will always be accessed by RPC with direct IO. Therefore, TeRM designs a dynamic hotspot

promotion mechanism, which relies on collaborative effort from clients and the server.

We implement TeRM by building a userspace library tLib with about 6,100 LoC, and modifying the Mellanox RNIC driver with about 300 LoC. tLib overrides the APIs of `libibverbs` and is compatible with existing RDMA applications. Using a microbenchmark, we demonstrate that TeRM achieves 98.13% throughput of the ideal upper bound with half physical memory. We also evaluate unmodified RDMA-based storage systems, a file system, Octopus [25], and a key-value system, XStore [37]. The results show that TeRM outperforms the ODP MR and the software-only RPC approach by up to 642.23× and 7.68×. We open source TeRM at https://github.com/thustorage/TeRM.

To sum up, we make the following contributions.

- We conduct an in-depth breakdown and analysis of the end-to-end latency to access the ODP MR.
- We propose TeRM, an efficient approach to extending RDMA-attached memory with SSD. It onloads exception handling (i.e., RNIC page fault) from hardware to software. We also introduce a set of techniques to reduce network traffic and CPU overhead.
- We use microbenchmarks and unmodified RDMA-based storage systems to demonstrate the effectiveness of TeRM.

2 Background

2.1 RDMA

RDMA registers and initializes two important resources on the control path, queue pair (QP) and memory region (MR). QP is the communication endpoint with another peer. MR exposes an area in the application's virtual memory, for the RNIC to access. Atop the initialized QP and MR, RDMA supports two types of requests on the data path, one-sided and two-sided. READ and WRITE are typical one-sided requests. RDMA applications leverage them to access data on a remote MR, without involving the remote CPU. SEND and RECV (receive) are typical two-sided requests that offer a message-passing abstraction. They are usually used to build RPC.

MR plays an indispensable role in freeing the remote CPU from being interrupted by a one-sided request. While initializing an MR, the driver pins all the pages in the physical memory, retrieves the virtual-to-physical mappings from the CPU page table, and stores them in the RNIC page table. We call the MR initialized in this way a *pinned MR* in the paper. With the pinned MR, the RNIC finds the physical addresses of the target virtual addresses in a one-sided request and accesses the data directly on the physical memory.

Although the pinned MR is prevalent in RDMA applications, it has several limitations. It pins a large number of pages in the physical memory (e.g., tens or hundreds of GBs), occupying valuable DRAM resources. The application can only initialize an MR no larger than the available physical memory.

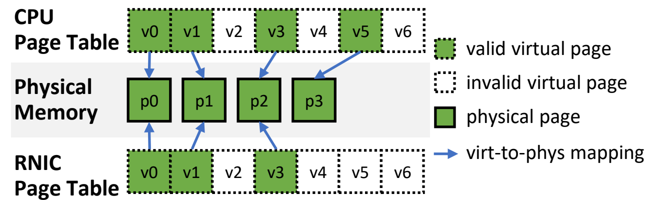


Figure 1: ODP MR. *We show the RNIC page table of an ODP MR and compare it with the CPU page table. A valid virtual page is mapped to a physical page in the page table. An invalid virtual page is not mapped. v5 is valid in the CPU page table but invalid in the RNIC page table. We explain the figure detailedly in §2.2.*

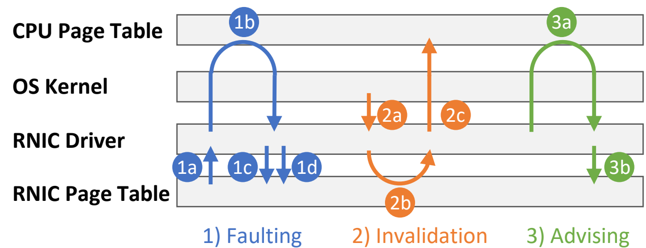


Figure 2: Flows to Synchronize CPU and RNIC Page Tables.

Meanwhile, it loses the opportunities for overcommitment, page migration, transparent huge-page, etc. Facing these limitations, ODP (On-Demand Paging) MR is proposed [22].

2.2 ODP MR

An ODP MR differs from a pinned MR in that it does not pin pages in the physical memory, as we depict in Figure 1. The RNIC page table maps some virtual pages to physical pages — we call them valid virtual pages — and leaves the rest unmapped, i.e., invalid virtual pages. Since the pages are no longer pinned, the OS kernel can swap and migrate pages. The application is able to expose an MR larger than the physical memory. As the virtual-to-physical mappings may be changed, CPU and RNIC page tables are synchronized by three flows illustrated in Figure 2.

1) Faulting. When an RDMA request accesses data on invalid virtual pages, (1a) the RNIC stalls the QP and raises an RNIC page fault[1] interrupt. (1b) The driver requests the OS kernel for virtual-to-physical mappings via `hmm_range_fault` [2]. The OS kernel triggers CPU page faults on these virtual pages and fills the CPU page table if necessary. (1c) The driver updates the mappings on the RNIC page table and (1d) resumes the QP.

2) Invalidation. When the OS kernel tries to unmap virtual

[1] we call the RNIC-triggered page fault an RNIC page fault in this paper, to distinguish it from a CPU page fault triggered by `load`/`store` in this paper

pages in scenarios like swapping out or page migration, (2a) it notifies the RNIC driver to invalidate virtual pages via `mmu_interval_notifier` [2]. (2b) The RNIC driver erases the virtual-to-physical mapping from the RNIC page table. (2c) The driver notifies the kernel that the physical pages are no longer used by the RNIC. Then, the OS kernel modifies the CPU page table and reuses the physical pages.

ODP MR relies on faulting and invalidation flows to synchronize CPU and RNIC page tables. All the valid virtual pages in the RNIC page table are guaranteed valid in the CPU page table, but not vice versa. When the kernel changes an invalid virtual page to a valid one, it does not inform the driver. As we illustrate in Figure 1, v5 is valid in the CPU page table but still left invalid in the RNIC page table.

3) Advising flow tackles the issue above. An application can proactively request the RNIC driver to populate a range in the RNIC page table. The RNIC driver completes advising by steps (3a) – (3b), which are identical to steps (1b) – (1c).

3 Motivation

In this section, we introduce RDMA-attached memory and analyze how ODP MR performs in extending it with SSD. We summarize two principles for designing TeRM.

3.1 RDMA-Attached Memory

An RDMA cluster includes several server and client machines that are equipped with RNICs and connected by RDMA network. By exposing the server's virtual memory with an MR, clients can directly read and write the RDMA-attached memory through RDMA READ/WRITE. Clients and servers may also exchange messages and data through RPC based on RDMA SEND/RECV. The RDMA-attached memory targets storage systems, e.g., file system and key-value system.

Note that the server's virtual memory is accessed both locally and remotely. Local accesses are from the CPU via `load`/`store`. Remote accesses are from clients via RDMA READ/WRITE. We take Octopus [25], an RDMA-based file system, as an example. The Octopus server initializes memory layout, maintains file metadata, and boots an RPC service for receiving and handling metadata requests. After retrieving file metadata (e.g., data addresses) via RPC, the Octopus client directly reads/writes the server-side MR to access file data.

As the server typically registers a pinned MR, it is restricted by the capacity of physical memory, as we explain in §2.1. To improve cost-efficiency and accommodate larger-than-memory data sets, we explore extending the RDMA-attached memory with SSD in this paper.

3.2 ODP MR Is Not the Silver Bullet

ODP MR enables a straightforward approach to extending RDMA-attached memory with SSD. The server-side application `mmap`-s an SSD to get a virtual memory area that exceeds the physical memory. Then it initializes an ODP MR for clients to access remotely.

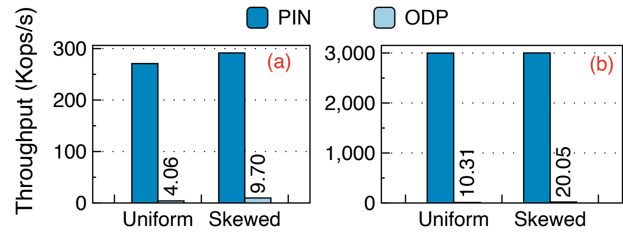


Figure 3: Read Throughput with (a) One Client Thread and (b) 64 Client Threads. *PIN: pinned MR. ODP: ODP MR. Read size: 4KB. MR size: 64GB. Physical memory for ODP MR: 32GB. SSD: Intel Optane P5800X. More detailed experimental setups are listed in §6.1.*

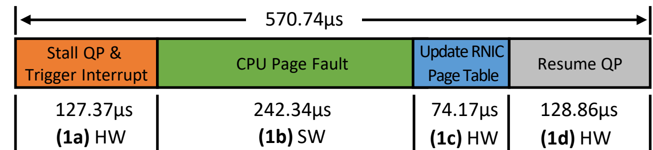


Figure 4: Time Breakdown of Read 4KB on an ODP MR. *(1a) – (1d): the steps introduced in Figure 2. HW: the step is executed by the RNIC hardware. SW: the step is executed by the software on the CPU.*

We use a microbenchmark to evaluate the performance of accessing a server-side ODP MR from the client. We `mmap` 64GB virtual memory from an Intel Optane P5800X SSD, and initialize a pinned MR (annotated as PIN) and an ODP MR (annotated as ODP) respectively; physical memory is limited to 32GB for the ODP MR. We evaluate the throughput of reading 4KB data on the MR. §6.1 offers more details about experimental setups. Figure 3 reports the results. The pinned MR outperforms the ODP MR by 66.64× with one client thread. The gap grows greatly to 290.76× with 64 client threads. The experiment shows that ODP MR exhibits poor performance, which is also reported by other works [16, 36]. Therefore, ODP MR is not the silver bullet for extending RDMA-attached with SSD.

We break down the end-to-end time to read 4KB on an ODP MR that triggers the RNIC page fault. Figure 4 depicts the time of four steps we introduce in §2.2. We do not draw the time of transferring 4KB data after resolving the RNIC page fault, because it occupies less than 5μs, which is negligible in the whole time. Notably, the CPU page fault (step(1b)) in our experiment is a major one where the OS kernel swaps data between the physical memory and the SSD, instead of a minor one [32]. We also evaluate the end-to-end latency with a minor page fault for comparison, which is 431.22μs. The latency difference stems from the software overhead of page

cache mechanisms to access the SSD, as the Intel Optane P5800X SSD has a read/write latency of only about 10μs.

The end-to-end time is composed of hardware time on the RNIC (steps (1a), (1c), and (1d)) and software time on the CPU (step (1b)). As shown in the figure, hardware steps take up more than half of the whole. When identifying an invalid virtual page during processing an RDMA request (step (1a)), the server-side RNIC returns a receiver-not-ready (RNR) negative acknowledgment packet (NACK) to the client-side RNIC [16, 22]. Then the QP is stalled on the request until it is resumed by step (1d). We presume that the latency of steps (1a) and (1d) arises from changing the QP state, which is reported to take up about 100μs [10, 40].

The root cause of the hardware's long latency is its inefficiency in handling exception cases. The limited compute and memory resources of the RNIC result in the simple approach of stalling the transmission. The RNIC circuitry of handling exception path operates relatively slowly, compared to the fast path of processing a normal RDMA request. Therefore, complex handling logic is too difficult to implement, as reported by researchers from Mellanox [22]. Given the above, we propose the first principle for designing TeRM. **Principle #1: onload exception handling from hardware to software.**

The other source of the end-to-end latency is software, the CPU page fault. The CPU page fault is known to perform poorly [11, 21, 28, 29] and does not scale well with the number of threads [30]. However, ODP MR makes the case even worse. During handling a CPU page fault, the kernel recycles a physical page, invalidates the virtual page mapped to it, and finally reuses it for the faulting virtual address. As we describe in §2.2, the kernel triggers the invalidation flow when invalidating a virtual page, where the driver spends considerable time updating the RNIC page table. Thus, the long latency of CPU page fault shown in Figure 4 implicitly includes invalidating the RNIC page table. Considering the above, we propose the second principle for designing TeRM. **Principle #2: eliminate CPU page faults from the critical path.**

4 Design

4.1 Overview

Figure 5 shows the overview of TeRM. We explain it below.

4.1.1 Architecture

Cluster infrastructure. The cluster has several servers and clients; we draw one server and one client in Figure 5 due to space limit. They are equipped with RNICs and connected via RDMA network. tLib is TeRM's userspace library (§5). It has two instances on the server (tLib-S) and the client (tLib-C).

CPU VM serves local access, i.e., CPU `load`/`store` from the server-side application. The server creates an area of virtual memory larger than the physical memory through `mmap`-ing

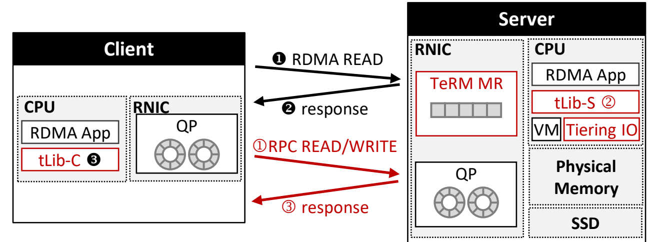


Figure 5: TeRM Overview. *Red ones are introduced by TeRM. tLib is TeRM's userspace library. We distinguish tLib on the client and the server by tLib-C and tLib-S respectively.*

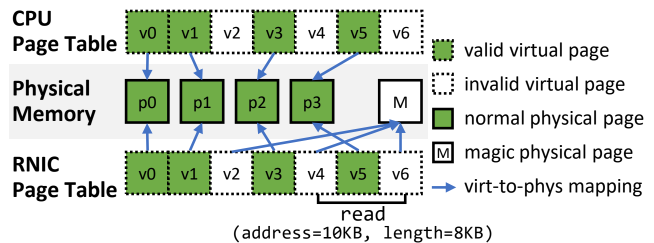


Figure 6: TeRM MR. *We show the RNIC page table of TeRM MR and compare it with the CPU page table. v0 – v6 are virtual pages. p0 – p3 and M are physical pages. The read request starts at the offset of 10KB with a length of 8KB.*

an SSD. TeRM leverages the Linux kernel to do the demand paging between the physical memory and the SSD, and manages the virtual-to-physical mappings in the CPU page table. In this way, the unmodified server-side application can access the virtual memory to maintain in-memory runtime data.

TeRM MR serves remote access from the client-side application. During initialization, the server-side application registers a TeRM MR to expose the virtual memory. tLib-S cooperates with the modified RNIC driver (§5) to manage the RNIC page table of the TeRM MR. Recall *principle #1: onload exception handling from hardware to software*. We orchestrate the RNIC page table and remove RNIC page faults, i.e., the faulting flow (Figure 2) from the TeRM MR.

As illustrated in Figure 6, For all the valid virtual pages (v0, v1, v3, v5), the RNIC page table maps them to normal physical pages (p0 – p3), the same ones that the CPU page table points to. When an RDMA READ accesses valid virtual pages, it retrieves the true data on the correct physical pages. For all the invalid virtual pages (v2, v4, v6), TeRM maps them to one magic physical page (M). TeRM reserves the magic physical page and populates it with a magic pattern. When an RDMA READ accesses invalid virtual pages, the server-side RNIC follows the mapping and retrieves the data on the magic physical page. In this way, the RDMA READ completes normally without triggering the RNIC page fault.

4.1.2 Workflow

Read. A client reads data on the server-side TeRM MR by submitting a read request to tLib-C. The read request describes the addresses of both sides and the length. In Figure 5, tLib-C processes the read request in three steps. ❶ tLib-C generates an RDMA READ according to the user-submitted read request and sends the RDMA READ via the client-side RNIC. ❷ The server-side RNIC returns the data without interacting with the CPU. ❸ tLib-C checks whether the data contains the predefined magic pattern. If no magic pattern is found, all the data are on valid virtual pages. tLib-C has retrieved the valid data and thus completes the read request. In this way, TeRM completes the read request by a one-sided RDMA READ.

If the magic pattern is found, the client determines that it accesses invalid virtual pages and data on these pages are missing. The client fetches the missing data in three steps. ① tLib-C submits an RPC to retrieve the missing data; we call it *RPC READ* hereinafter. ② Receiving an RPC READ, tLib-S reads the missing data to a preallocated and registered bounce buffer. ③ tLib-S returns data to the client. As the bounce buffer has been registered, tLib-S sends the data via RDMA WRITE without triggering RNIC page faults. Afterward, the server notifies the client of the completion. Finally, with all the data fetched, tLib-C completes the read request.

Write. A client writes data to the server-side TeRM MR by submitting a write request to tLib-C. tLib-C processes the write request in three steps like processing a missing read request. ① tLib-C submits an RPC to the server to write the data; we call it *RPC WRITE* hereinafter. ② tLib-S fetches the data from the client to the bounce buffer by RDMA READ. Then it copies the data on the bounce buffer to the virtual memory. ③ tLib-S notifies the client, and tLib-C completes the write request to the application.

4.1.3 Challenges

There are several challenges in the design.

1) As a read request is in byte granularity but the virtual-to-physical mapping of the TeRM MR is in page granularity, precisely identifying invalid virtual pages becomes challenging. We tackle the challenge in a hierarchical manner, from the request level to the page level. We also introduce a set of techniques to reduce network traffic during identification. We detail the design in §4.2.

2) As mentioned in the workflow, TeRM may introduce internal RPCs, i.e., RPC READ and RPC WRITE that access the SSD-extended virtual memory. An intuitive approach is performing `load`/`store`, inducing heavy CPU page faults. Following *principle #2: eliminate CPU page faults from the critical path*, we propose tiering IO. Instead of memory `load`/`store` interfaces, tiering IO resorts to file IO interfaces, i.e., selectively uses buffer IO and direct IO to access the SSD-extended virtual memory. We describe how tiering IO operates at length in §4.3.

3) As TeRM MR and tiering IO eliminate RNIC and CPU page faults from the critical path, it freezes the data placement on the server, unfortunately. A fixed part of the virtual memory is in the physical memory and mapped in the TeRM MR. Without promotion on the critical path by the page faults, a hotspot may be always on the SSD. Facing the challenge, TeRM makes the client and the server collaborate to determine and promote hotspots to physical memory in the background dynamically (§4.4).

4.2 Identifying Invalid Virtual Pages

As a read request is in byte granularity, it leads to two issues during identifying invalid virtual pages, the inter-page issue at the request level and the intra-page issue at the page level. The inter-page issue is that a read request may span multiple virtual pages, some of which are valid but others are invalid. For efficiency, TeRM should identify and fetch only the missing data on invalid virtual pages via RPC. The intra-page issue is that a read request may access only part of a virtual page. TeRM should be able to determine whether a virtual page is valid with any part of the virtual page. Moreover, TeRM should reduce network traffic in identification.

Page division. To tackle the inter-page issue, TeRM adopts page division. It splits the received data at page boundaries into several parts and checks each part separately. Take the read request in Figure 6 as an example. TeRM cuts the data into three parts (on v4 – v6) and checks them one by one.

Byte detection. To tackle the intra-page issue, TeRM adopts byte detection. On the server side, the magic pattern covers all the bytes — not just the beginning or the end — of the magic physical page, e.g., setting every byte to a magic number. On the client side, tLib-C compares the retrieved part of a virtual page with the magic pattern byte by byte. If matched, tLib-C assumes that the part belongs to an invalid virtual page. The read request in Figure 6 accesses the first half of v4 and the last half of v6, which match the magic pattern, so tLib-C determines v4 and v6 are invalid. The data on v5 does not match the magic pattern and thus v5 is valid.

Sparse fetching. After identifying all the invalid virtual pages precisely, tLib-C fetches data sparsely. It submits an RPC READ and tLib-S fetches only the missing data on them. Apart from the addresses of both sides and the access length, the RPC READ also contains a page bitmap to indicate whether each page is valid. With the read request in Figure 6 as an example, the page bitmap is `b'010`, as the first (v4) and the last (v6) virtual pages are invalid. Receiving the RPC READ, tLib-S bypasses all the valid virtual pages. It parses the bitmap to locate and read all the invalid virtual pages.

Combining page division, byte detection, and sparse fetching, TeRM identifies invalid virtual pages of a read request and only fetches the missing data on these pages via RPC. Compared with fetching all the data of a read request, of which only some virtual pages are invalid, TeRM reduces the

amount of data transfer and thus speeds up the miss path.

False positive cases. Identifying invalid virtual pages by the magic pattern may lead to false positive cases. If the server-side application fills a valid virtual page with the magic pattern, the client will determine it as an invalid virtual page falsely. TeRM overcomes the issue with three key insights. First, for random data, the possibility of false positive cases is low. For a single byte (i.e. 8 bits) of random data, the probability is only $1/2^8 = 1/256$. As the data length grows to n bytes, the probability drops exponentially to $1/256^n$, which is negligible. Moreover, TeRM varies the magic pattern dynamically for different processes at different times, to prevent an application from always producing the same data as one specific magic pattern. Finally, even if a false positive case occurs, TeRM handles it as accessing an invalid virtual page and fetches the data again via an RPC READ, without compromising the correctness.

Page bitmap. As tLib-C identifies an invalid virtual page from the magic pattern, RDMA READ for it consumes extra network bandwidth. To reduce the network traffic, we propose a page bitmap to identify an invalid virtual page before RDMA READ. tLib-S maintains a page bitmap for a TeRM MR to indicate whether each virtual page is valid. tLib-C pulls the page bitmap periodically, e.g., per second in our evaluation. For a read request, tLib-C queries the page bitmap first and only sends RDMA READ for valid virtual pages. Afterward, tLib-C submits RPC READ for all invalid virtual pages identified by both the page bitmap and the magic pattern.

Note that the client-side page bitmap may be inaccurate but does not harm read correctness and the overhead is acceptable. If an invalid virtual page is indicated as valid by the bitmap, RDMA READ will return the magic pattern and thus tLib-C can identify it correctly. In contrast, for a valid virtual page indicated as invalid, RPC READ will retrieve the correct data.

One may wonder why we do not use the page bitmap to guide a write request, i.e.,sending RDMA WRITE for a valid virtual page and RPC WRITE for an invalid one. This is because the overhead due to inaccuracy is unacceptable. If an invalid virtual page is indicated as valid, RDMA WRITE on it will trigger an RNIC page fault, which stalls the transmission and consumes no less time than a read-triggered one (hundreds of microseconds as shown in Figure 4).

We discuss the overhead of pulling the page bitmap. With one bit for each 4KB page, the page bitmap size is only 0.003% of the MR. For a 64GB MR, each client pulls 2MB each time, which is negligible against the RNIC bandwidth.

Although TeRM introduces extra network traffic in identifying an invalid virtual page, we argue that the ODP MR also causes additional network traffic due to the RNR NACK. It stalls the QP and wastes more network resources (§3.2).

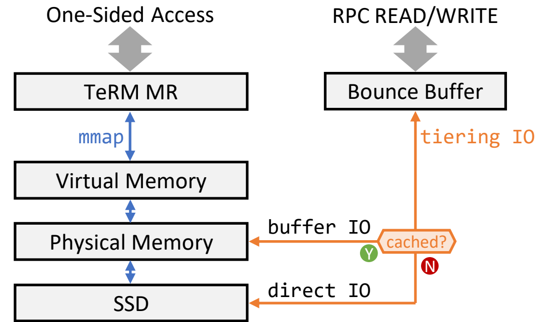


Figure 7: Tiering IO.

4.3 Accessing Data via Tiering IO

As we state in the workflow (§4.1), tLib-S reads and writes the virtual memory to/from the bounce buffer during handling RPC READ/WRITE. Recall that TeRM `mmap`s an SSD to create an area of virtual memory, so that the unmodified server-side application on the CPU can `load`/`store` the SSD-extended virtual memory. Therefore, tLib-S must also access the virtual memory through a kernel-exposed interface, instead of a kernel-unaware interface, e.g., SPDK [7].

Although `memcpy` between the virtual memory and the bounce buffer is an intuitive choice, it triggers heavy CPU page faults (Figure 4) on invalid virtual pages that have been swapped out to the SSD. Following *principle #2: eliminate CPU page faults from the critical path* (§3.2), we propose tiering IO to access the SSD-extended virtual memory, as illustrated in Figure 7. Our key idea is resorting to file IO interfaces instead of memory `load`/`store` interfaces.

Tiering IO orchestrates two interfaces — buffer IO and direct IO — to access different states of virtual pages. Buffer IO invokes `pread`/`pwrite` to access the page cache. Direct IO bypasses the page cache with the `O_DIRECT` flag.

Tiering IO selects the interface to access a virtual page according to its state. 1) If the virtual page resides in the page cache, tiering IO accesses it via buffer IO. The IO can be completed fast by the page cache, without communicating with the SSD. 2) If the virtual page is uncached, accessing the data via buffer IO will incur the page replacement of the page cache. The replacement is time-consuming [5, 8], especially when the page cache is nearly full. Therefore, tiering IO chooses direct IO to bypass the page cache.

We identify three issues to support tiering IO, virtual-to-block mapping, virtual page state, and direct IO granularity. We discuss and tackle them below.

Virtual-to-block mapping. RPC READ/WRITE provides the virtual address to access, we have to convert it to a logical block address (LBA) on SSD for invoking IO. Fortunately, the Linux kernel offers an efficient static virtual-to-block mapping. By `mmap`-ing a given LBA range `[slba, slba + length)` of the SSD, we get a virtual address range `[saddr,`

`saddr + length)`. For a server-side virtual address `addr`, tLib-S calculates its LBA by `addr - saddr + slba`.

Virtual page state. TeRM queries the page cache to learn whether a page is cached. Notably, the page state may be stale by the time invoking the IO call. For example, tiering IO determines that a virtual page is uncached and then accesses it via direct IO, but the page may be cached just before the call begins. We argue that the stale page state does not compromise the correctness, because direct IO read and write flushes and invalidates the page cache respectively, so as to guarantee data consistency [1].

Direct IO granularity. The granularity of RPC READ-/WRITE and direct IO does not match. The former is a byte, while the latter is a block, typically 512B or 4KB. To bridge the granularity gap for RPC READ, we pad offset and length of `pread` to block boundaries. As for RPC WRITE unaligned to a block, we adopt a read-modify-write operation. We use an exclusive lock for each block to control the concurrent read-modify-write operations on the same block.

4.4 Determining and Promoting Hotspots

With the design of TeRM MR and tiering IO, TeRM eliminates RNIC and CPU page faults form the critical path. Although the elimination streamlines the critical path, it freezes the data placement on the server, unfortunately. A fixed part of the virtual memory is in the physical memory and mapped in the TeRM MR. If a hotspot is on the SSD, it will always be accessed by an RPC READ/WRITE with direct IO. Considering the server is unaware of one-sided RDMA accesses from the client, we propose making the client and the server collaborate to determine hotspots and then promote hotspots dynamically, so as to improve the overall performance.

Determining hotspots. TeRM employs client-side tracking and server-side accumulating to count the frequency of read-/write requests and find the hotspots.

TeRM tracks requests at the client, because a hit read request finishes by a one-sided RDMA READ without involving the server-side CPU. tLib-C splits the address space of a TeRM MR at the granularity of a *sample unit* and creates a counter for each unit. When the application submits a read-/write request, tLib-C locates all the requests' spanning sample units and increases their counters. A smaller sample unit results in finer counting, but the TeRM MR is divided into more units and thus the counters occupy a larger memory space. TeRM sets the sample unit to 1MB to achieve the balance between the counting granularity and the counters' memory footprint. With a 32-bit counter for each sample unit, the counters take up only 0.00003% memory space compared to a TeRM MR, which is negligible.

TeRM accumulates counters at the server, given that multiple clients in the cluster may access one TeRM MR. In every *sample period*, tLib-S pulls all the counters from the clients and sums them up. Then it gets a global view of the counters and knows how many times each unit has been accessed in the latest period. A shorter sample period leads to a more timely counting but consumes more network bandwidth during transferring the counters. TeRM sets the sample period to 1 second to balance these two aspects.

The more times that a sample unit is accessed, the hotter TeRM thinks it is. TeRM sorts the units by their counters in descending order and determines the hottest units that can be placed in the physical memory as hotspots.

Promoting hotspots. TeRM promotes hotspots one by one. A unit is skipped if it has been promoted in an earlier period. Otherwise, TeRM invokes the advising flow (Figure 2) to promote the unit. The unit is swapped into the physical memory and mapped in the RNIC page table. Then a later read request on the unit is completed via an RDMA READ and a write one is done via buffer IO write in RPC WRITE.

The advising flow is also time-consuming due to triggering the CPU page fault and updating the RNIC page table. Thus, TeRM does not promote all the hotspots in one promotion. Instead, TeRM promotes as many units as possible within the time of a sample period. Then it begins the next period of determining and promoting hotspots.

The promotion design balances effectiveness and flexibility. If the hotspots remain stable for consecutive periods, TeRM promotes the hottest proportion in the beginning periods and then the less hot ones in the later periods. All the determined hotspots are promoted eventually. However, if the hotspots have changed since the last sample period, promotion for the last period completes fast, and TeRM shifts to promote the latest hotspots immediately. As the promotion is conducted periodically, it occupies little CPU resources.

Consistency discussion. As the promotion and tiering IO in RPC READ/WRITE may access the same page, we discuss the concurrency consistency here. As both trap into the Linux kernel, TeRM reuses the concurrency control in the Linux kernel to guarantee consistency. Note that the promotion always accesses the page cache. If tiering IO performs buffer IO, accesses are routed to the page cache, and hence the concurrency consistency is maintained by the page cache. If tiering IO performs direct IO, read requests do not raise any consistency issue since the SSD always contains the newest version of data. At the beginning and end of write requests in direct IO, the kernel invalidates the page cache so as to prevent old data in the cache and guarantees the consistency.

5 Implementation

We implement TeRM for the Mellanox RNIC. We build the userspace library tLib with about 6,100 lines of C++ code and modify the RNIC driver with about 300 lines of C code.

tLib. It overrides the APIs to manipulate the MR (`ibv_register_mr`) and the RDMA request (`ibv_post_send`, `ibv_poll_cq`). tLib is transparent

to the upper-layer application via `LD_PRELOAD` and has two instances tLib-S and tLib-C.

The server-side application invokes `ibv_register_mr` to register an MR. In the overridden `ibv_register_mr`, tLib-S interacts with the modified RNIC driver to create a TeRM MR and starts an RPC service in the userspace to serve the RPC READ/WRITE from clients. We adopt coroutine in the RPC service and `libaio` to submit direct IO to the SSD asynchronously, so as to enhance the CPU efficiency.

The client-side application calls `ibv_post_send` to submit a read/write request and polls the completion via `ibv_poll_cq`. For a write request, tLib-C converts it to an RPC WRITE in the overridden `ibv_post_send` and submits it to the server. For a read request, tLib-C identifies invalid virtual pages in the overridden `ibv_poll_cq` and submits an RPC READ to the server if necessary.

Considering RDMA requests enjoy low latency, we aim to make tLib execute efficiently. We employ multithreading-friendly and cacheline-aware data structures and mechanisms throughout the implementation, to reduce the extra running overhead introduced by tLib.

RNIC driver. We modify the RNIC driver to support the TeRM MR. We reuse the mechanisms of the ODP MR, including the RNIC page table and the synchronization flows with the CPU page table§2.2. When creating a TeRM MR, the RNIC driver allocates a physical page from the kernel and fills it with the magic pattern. TeRM eliminates the faulting flow as we state in §4 and modifies the invalidation flow in the driver. When the Linux kernel notifies the RNIC driver of invalidating a virtual page, the RNIC driver does not clear the virtual-to-physical mapping as it does for the ODP MR, but instead makes the mapping point to the magic physical page.

6 Evaluation

We evaluate TeRM by microbenchmarks and RDMA-based storage systems to answer the following questions.

- How does TeRM compare with existing approaches? (§6.2)
- How do the design techniques contribute to the end-to-end performance of TeRM? (§6.3)
- How does TeRM perform on dynamic workloads? (§6.4)
- How do workload characteristics affect TeRM? (§6.5)
- How can RDMA-based storage systems benefit from TeRM? (§6.6)

6.1 Experimental Setup

Testbed. We conduct the experiments on a cluster of one server machine and two client machines. The server machine has a 56-core Intel Xeon Gold 6330 CPU, 96GB DRAM, and a 400GB Intel Optane 5800X SSD. The SSD has 1.25/1.16 Mops/s of 4KB random read/write and 4.21/0.69 Mops/s of 512B random read/write. Each client machine has a 36-core Intel Xeon Gold 5220 CPU and 64GB DRAM. We equip the machines with a ConnectX-5 RNIC on each and connect them by a 100Gbps IB RDMA switch.

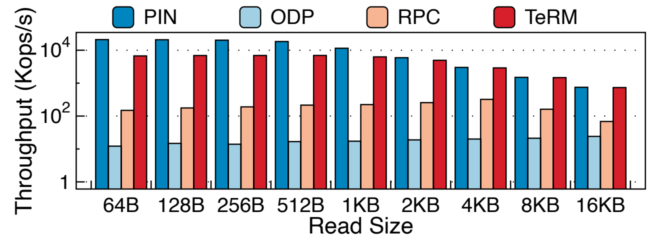


Figure 8: Read Throughput. *The vertical axis is in a logarithmic scale.*

Comparing Targets. We compare TeRM with two approaches ODP and RPC. Moreover, we use PIN to show the ideal upper bound of performance where all data pages are pinned in the physical memory.

- *PIN*. Only in this approach, we do not restrict the available physical memory. The server registers the virtual memory as a pinned MR. The clients read and write the server-side pinned MR by one-sided RDMA READ/WRITE through the original `libibverbs`.
- *ODP*. On the server machine, we register the virtual memory as an ODP MR. The clients also use the original `libibverbs` to submit read/write requests. All the requests are handled by one-sided RDMA READ/WRITE and trigger RNIC page faults on invalid virtual pages.
- *RPC*. All the read/write requests are handled by RPC READ/WRITE. The server-side RPC service accesses the virtual memory via `memcpy` and thus triggers CPU page faults when a virtual page is not mapped.
- *TeRM*. The server registers the virtual memory as a TeRM MR. tLib-C interacts with tLib-S to handle read/write requests submitted by the client, as we describe in the design and implementation.

Workloads. We use a microbenchmark to evaluate the performance. We run it for 60 seconds and report the average throughput. It creates a 64GB virtual memory by `mmap`-ing the SSD on the server machine. We limit the available physical memory to 32GB, 50% size of the virtual memory. The microbenchmark runs 64 client threads, 32 threads on each client machine. Each client thread issues read and write requests to the server, where the accessing positions follow a skewed distribution (Zipfian θ=0.99). For both the RPC approach and TeRM, we create 16 threads for the RPC service on the server machine and bind these threads on eight physical CPU cores. We keep the settings above as the default throughout the experiments unless stated otherwise.

6.2 Overall Performance

In this experiment, we evaluate the read and write performance of access sizes from 64B to 16KB.

	Read 256B		Read 4KB	
	p50 lat.	**p99 lat.**	**p50 lat.**	**p99 lat.**
PIN	3.10	4.21	21.20	24.03
ODP	3.03	29,103.92	4.60	45,781.38
RPC	38.55	109.79	26.01	93.62
TeRM	3.50	30.07	16.51	52.02

Table 1: Read Latency (μs)

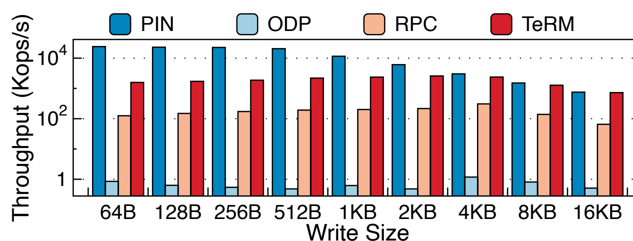


Figure 9: Write Throughput.*The vertical axis is in a logarithmic scale.*

	Write 256B		Write 4KB	
	p50 lat.	**p99 lat.**	**p50 lat.**	**p99 lat.**
PIN	2.91	3.89	21.29	23.40
ODP	56,158.64	71,006.89	19,884.87	41,636.82
RPC	43.89	131.52	317.12	1,772.35
TeRM	15.22	117.27	21.18	59.73

Table 2: Write Latency (μs)

6.2.1 Read

We report read throughput in Figure 8 and latency in Table 1. We analyze the performance of TeRM against ODP, RPC, and PIN respectively in the following.

TeRM vs. ODP. The throughput of TeRM and ODP achieves 6.93Mops/s and 24.09Kops/s respectively. TeRM outperforms ODP by 30.46× – 549.63×. ODP has the lowest p50 latency of all four approaches. This is because most read requests are hit in the physical memory. The RNIC on the ODP approach is the least utilized in transferring data and thus shows the lowest latency to finish a hit RDMA READ. However, the long p99 latency demonstrates that ODP suffers from heavy RNIC and CPU page faults on the miss path. TeRM reduces the p99 latency by up to 967.74×, thanks to the much more efficient miss path.

Notably, PART [32], a hardware solution like the ODP MR, reports a 31μs latency of a faulting RDMA request, which is lower than ODP in our experiment. The latency difference mainly arises from the fact that PART evaluates a minor page fault while we evaluate a major one. A major page fault has to load data from the SSD. Nevertheless, TeRM incurs a lower average latency of 26.61μs for a missed read request. This is because TeRM leveragers tiering IO to build an efficient miss path that bypasses the CPU page fault.

TeRM vs. RPC. The RPC approach reaches a throughput of 320.75Kops/s. It does not provide the maximum throughput, because the RPC approach triggers CPU page faults when pages are not mapped. The CPU page fault performs poorly and does not scale well with more server threads, which is also reported by previous studies [11, 21, 28–30]. The RPC approach performs better than ODP because clients use two-sided RDMA primitives to submit read requests. In this way, the RPC approach avoids RNIC page faults. TeRM surpasses the throughput of the RPC approach by 9.05× – 45.19×. It decreases the p50 and p99 latency by 11.02× and 1.57× respectively. TeRM has significant improvement because it utilizes the server CPU more efficiently in two aspects. First, TeRM tries one-sided RDMA READ and only handles the missed read requests via the RPC service on the server CPU, but the RPC approach involves the server CPU in processing all the read requests. Moreover, TeRM proposes tiering IO to avoid CPU page faults, while the RPC approach may still trigger CPU page faults during memcpy.

TeRM vs. PIN. When the read size is less than 1KB, the throughput of TeRM is stable around 6.86Mops/s, achieving up to 37.79% of the PIN throughput. With 256B as an example, the hit ratio of read requests is 69.44%, and the latency is as low as PIN. The slowdown of TeRM mainly arises from the missed read requests, each of which costs about 19.26μs. In this scenario, the RPC service becomes the bottleneck of the overall performance.

For large read requests above 1KB, TeRM greatly narrows the gap with PIN. It achieves 54.55% throughput of PIN at 1KB and the ratio goes up to 96.71% at 4KB. The narrowing gap results from the shrinking latency difference between hit (16.51μs) and missed (26.61μs) read requests. In this case, TeRM saturates the RNIC bandwidth.

6.2.2 Write

We show write throughput in Figure 9 and latency in Table 2. We compare TeRM with ODP, RPC, and PIN.

TeRM vs. ODP. TeRM and ODP have throughput up to 2.58Mops/s and 1.18Kops/s respectively. We output the throughput second by second and find that ODP is unstable and jitters sharply. It reaches a peak throughput of 4.28Kops/s at some time but may also stall for more than a second. Nevertheless, TeRM surpasses ODP's peak throughput by up to 1,195.81×. ODP performs worse on write than read, because write incurs higher swapping overhead. To swap out a read-only page, the OS kernel just drops it from the physical memory because it is clean. But to swap out a written page, the OS kernel has to write the dirty page back to the SSD.

TeRM vs. RPC. The RPC approach reaches a throughput of 308.34Kops/s. Even though TeRM also handles write requests via RPC WRITE, it outperforms the RPC approach by up to 12.60×. TeRM reduces the p50 and p99 latency by 14.97× and 29.67×. The results demonstrate the effectiveness of tiering IO and promoting hotspots compared to `memcpy`.

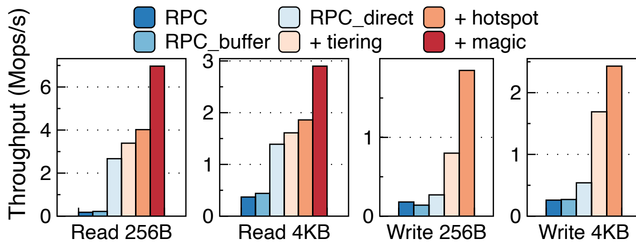


Figure 10: Contribution of Each Technique.

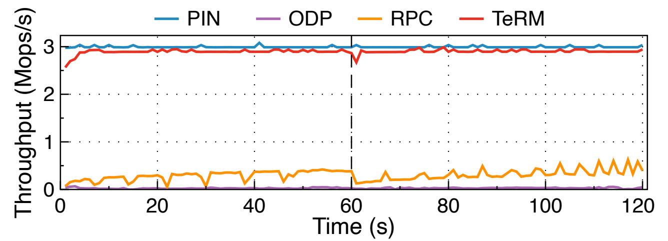


Figure 11: Performance of Dynamic Workloads. *We change the hotspots at the 60th second.*

TeRM vs. PIN. When the write size is smaller than 512B, the throughput is around 1.71Mops/s. Taking 256B as an example, 72.53% of write requests hit the page cache and finished by buffer IO. The rest of write requests are completed by the read-modify-write operation as we describe in §4.3, which limits the overall performance. When the size grows to 512B, the write throughput climbs to 2.19Mops/s. This is because tiering IO writes uncached data simply by a direct IO write rather than the time-consuming read-modify-write. The SSD can provide 694Kops/s of 512B write operations and becomes the bottleneck. As the write size goes up, the SSD's throughput increases and so does the write throughput of TeRM. TeRM reaches the throughput of 2.38Mops/s at 4KB, 78.69% of the PIN throughput, where the SSD exhibits 1.16Mops/s of 4KB write. When the write size is even larger, TeRM further reduces its gap with the PIN approach. TeRM achieves 727.28Kops/s throughput at 16KB, 96.32% of the PIN approach. In this scenario, the RNIC bandwidth dominates the overall performance.

6.3 Contribution of Each Technique

In this experiment, we analyze how each technique contributes to TeRM, as reported in Figure 10. We choose 256B and 4KB as representatives of small and large read/write sizes. We use three baselines, RPC, *RPC_buffer*, and *RPC_direct*. RPC_buffer and RPC_direct are the same as the RPC approach, except that we replace the server-side `memcpy` with buffer IO and direct IO respectively. We introduce these two baselines to study the advantage of tiering IO compared with existing IO interfaces. Then we gradually enable tiering IO (§4.3), promoting hotspots (§4.4), and TeRM MR (§4.1&§4.2) atop RPC_direct. Three techniques are annotated as *+tiering*, *+hotspot*, and *+magic* in Figure 10. We test Read 4KB using eight RPC threads, which are enough for TeRM in this case; we analyze server-side CPU usage in more detail later in §6.5.4. We apply the TeRM MR last because it can function better when hotspots are promoted. Notably, the TeRM MR does not apply to write requests.

The experimental results demonstrate that all techniques contribute to the performance improvement of TeRM.

Baselines. We first compare three baselines. RPC_buffer performs as poorly as RPC. It avoids CPU page faults but cannot eschew the heavy page replacement [5, 8]. RPC_direct surpasses RPC_buffer by 1.90× – 12.05×. It bypasses the page cache but loses the opportunity to access cached data fast.

+tiering. Tiering IO improves the performance by 1.16× – 3.10× over RPC_direct. It accesses data via the page cache whenever possible. When the data is on the SSD, tiering IO accesses it directly through the device. In this way, tiering IO manages to exploit the high-performance physical memory and avoid the heavy page cache maintenance simultaneously.

+hotspot. Determining and promoting hotspots further increases the throughput by 1.16× – 2.31×. With the hotspots promoted in the physical memory, tiering IO completes more hot data requests from the page cache.

+magic. TeRM MR raises the throughput by 1.56× – 1.73×. The hit read requests are handled through one RDMA READ operation without bothering the server-side CPU. As the RPC service only handles miss read requests instead of all read requests, TeRM utilizes both the RNIC and the server-side CPU more efficiently.

More specifically, the page bitmap also plays a remarkable role in performance improvement. The hit ratio of read 4KB requests is about 73%. Without the page bitmap, the remaining 27% read requests are transferred twice, the first time via RDMA READ and the second time via RPC READ. The end-to-end throughput is 2.37Mops/s, only 78.97% of the PIN approach. With the page bitmap, less than 0.1% read requests are transferred twice. The throughput is 2.90Mops/s, achieving 96.71% of the PIN approach.

6.4 Dynamic Workloads

We evaluate how TeRM reacts to dynamic hotspots and plot the results in Figure 11. We run the benchmark for 120 seconds and change the hotspots at the 60th second. We have two observations from the results. 1) TeRM performs more stably than ODP and RPC. TeRM is stable at 2.89Mops/s and drops by only 6.82% after switching hotspots. The throughput of ODP and RPC jitters sharply and drops by 1.77× and 3.03× after the switching. 2) TeRM determines and promotes hotspots effectively and efficiently. The throughput of TeRM returns to the peak quickly in one second, while it takes ODP

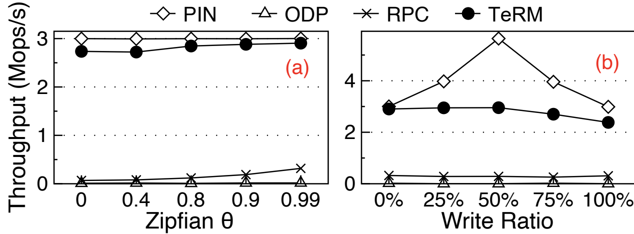


Figure 12: Performance with Varying (a) Skewness and (b) Write Ratios.

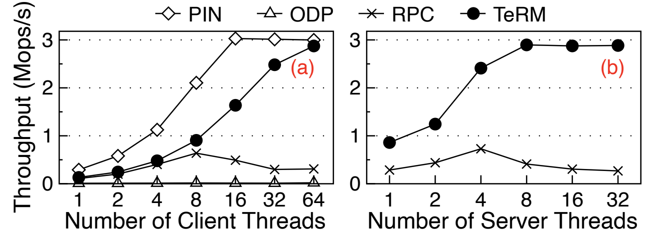


Figure 13: Performance with Different Numbers of (a) Client Threads and (b) Server Threads.

and RPC six seconds.

6.5 Sensitivity Analysis

We evaluate how the characteristics of workloads impact the performance of TeRM. We show the read performance of 4KB in these experiments by default, unless otherwise stated.

6.5.1 Skewness

Figure 12(a) plots the performance of approaches with varying skewness. For the uniform distribution ($\theta = 0$), TeRM exhibits more significant improvement against existing approaches, compared with the skewed distribution. It outperforms ODP and RPC by 265.24× and 40.40× respectively, achieving 91.22% of PIN. The hotspots are more concentrated when θ increases. The PIN approach is stable with varying skewness. ODP, RPC, and TeRM show higher throughput as the skewness grows, because more requests are within the hotspots that reside in the physical memory.

6.5.2 Write Ratio

Figure 12(b) depicts the throughput with five different write ratios, 0% (read-only), 25% (read-most), 50% (read-write), 75% (write-most), and 100% (write-only). The PIN approach shows improvement on read-write-mixed requests because it exploits the full-duplex performance of the RDMA network. As for TeRM, mixing read and write slows down the direct IO performance and restricts the overall throughput. ODP and RPC do not perform better for read-write-mixed requests compared with the read-only or write-only scenario.

6.5.3 Client Threads

Figure 13(a) reports the throughput with different numbers of client threads. As the number of client threads goes up, the throughput of TeRM grows linearly and reaches 2.48Mops/s at 32 threads. The throughput of PIN also increases linearly and hits the peak at 16 threads. ODP and RPC scale poorly with the increasing number of client threads.

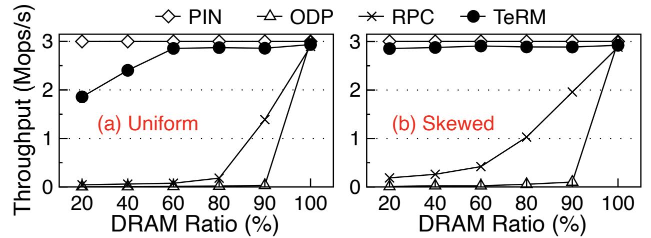


Figure 14: Performance with Varying DRAM Ratios of (a) Uniform and (b) Skewed Workloads.

6.5.4 Server Threads

Figure 13(b) shows the throughput with different numbers of server threads for the RPC service. TeRM scales with the increasing number of server threads and reaches the peak at eight threads. The RPC approach does not scale well because the CPU page fault scales poorly [30]. TeRM outperforms the RPC approach by 2.84×– 10.76×. Even with one server thread, TeRM exceeds the peak throughput of the RPC approach. The results demonstrate the CPU efficiency of TeRM.

6.5.5 DRAM Ratio

As shown in Figure 14, we evaluate the performance with different sizes of DRAM, i.e., available physical memory. TeRM performs well with varying DRAM sizes. Even with only 20% DRAM, TeRM provides 95.10% and 61.93% throughput of the PIN approach on skewed and uniform workloads. It outperforms ODP and RPC by up to 388.29× and 41.78×. The enhancement is higher compared with the default 50% DRAM setting in our experiments. This demonstrates that TeRM still acts efficiently under a low DRAM ratio. All approaches have higher throughput with more DRAM. With the 90% DRAM ratio, the RPC approach increases to 1.95Mops/s.

It is worth noting the performance with a 100% DRAM ratio, where all data fits in the physical memory. This scenario shows the extra overhead of each approach. Compared with PIN, TeRM introduces 2.63% overhead, which is negligible. ODP and RPC also exhibit performance close to PIN.

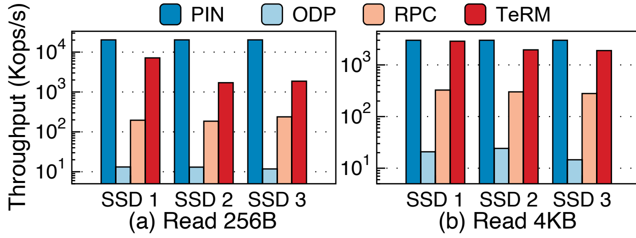


Figure 15: Performance with Different SSDs of (a) Read 256B and (b) Read 4KB. *The vertical axis is in a logarithmic scale. SSD 1: Intel Optane P5800X. SSD 2: Intel Optane P4800X. SSD 3: Samsung PM9A3. More details about SSDs are listed in Table 3.*

ID	Product Model	Read 512B	Read 4KB
SSD 1	Intel Optane P5800X [4]	4,216	1,255
SSD 2	Intel Optane P4800X [3]	586	586
SSD 3	Samsung PM9A3 [6]	600	619

Table 3: Throughput of Different SSDs (Kops/s). *We test their random throughput on a 64GB area using `fio` with 16 threads and libaio (queue depth = 4). SSD 1 & 2 use Intel Optane memory as the storage media. SSD 3 uses NAND flash as the storage media.*

6.5.6 SSD

We evaluate how different SSDs impact the performance and plot the results in Figure 15. The details of the SSDs are listed in Table 3. TeRM running on SSD 2 and SSD 3 achieves close throughput at about 1.95Mops/s. It outperforms ODP and RPC by up to 158.83× and 9.22×. SSD 2 and SSD 3 have similar IO throughput and limit the overall performance. The experimental results show that TeRM acts effectively on different SSDs with different types of storage media.

6.6 RDMA-based Storage Systems

We evaluate how existing RDMA-based storage systems can benefit from TeRM. We choose an RDMA-based file system, Octopus [25], and an RDMA-based key-value system, XStore [37]. We keep the programs unmodified, except `mmap`-ing the SSD to get a large area of virtual memory and registering it as a pinned, ODP, or TeRM MR in their initialization stage.

6.6.1 Octopus: A File System

Octopus is an RDMA-based file system. The server initializes a large area of virtual memory to store metadata and data, and exposes it via an MR. Meanwhile, it runs an RPC service for processing metadata. During accessing a file, the client first communicates with the server via the metadata RPC, to retrieve metadata of a file, e.g., data addresses. Then it reads or writes the server-exposed MR to access file data.

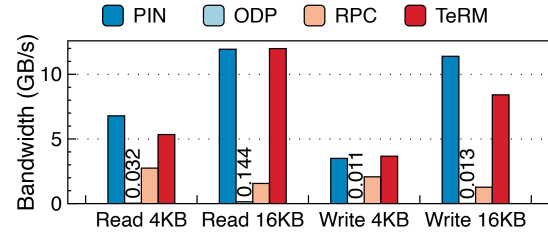


Figure 16: Octopus Performance.

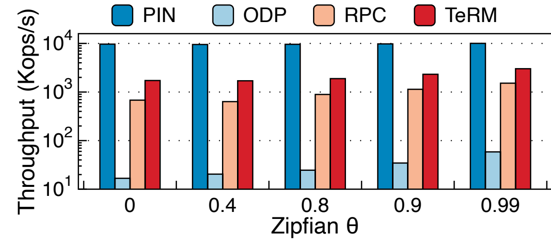


Figure 17: XStore Performance. *The vertical axis is in a logarithmic scale. We use the YCSB-C workload with 8B keys and 128B values.*

On the server machine, we boot 16 threads for metadata service on 16 cores. We run 32 client processes. Each of them reads/writes 4KB/16KB on a 1GB file, where the access positions follow a skewed distribution (Zipfian $\theta = 0.99$). The metadata and data occupy about 35GB of virtual memory on the server; we limit the available physical memory to 18GB.

Figure 16 reports the results. TeRM achieves 82.99–642.23× ODP and 1.77–7.68× RPC. It performs almost the same as PIN on Read 16KB and Write 4KB. Accessing 4KB is slower than 16KB because the client fetches metadata before transferring data. In this scenario, the metadata service bottlenecks the throughput.

6.6.2 XStore: A Key-Value System

XStore is an RDMA-based key-value system. The server maintains a B+ tree and trains a learned index on the virtual memory. It exposes the virtual memory via an MR. The client leverages the learned index to predict the value's address and reads the server-side MR to `get` it. XStore handles `put` operations via RPC. The server runs an RPC service to process `put` requests from the client.

In our experiment, the server initializes a B+ tree containing 8B keys and 128B values. XStore occupies 32GB of virtual memory on the server and we limit the available physical memory to 16GB. Since `put` operations are based on XStore's own RPC, we evaluate how TeRM benefits the `get` performance. We use a YCSB-C workload and vary the skewness of the keys' distribution.

Figure 17 shows the experimental results. TeRM outperforms the ODP and RPC approach by up to 102.97× and 2.69× respectively. As the skewness increases, `get` through-

put increases because hotspots are more concentrated in the physical memory. TeRM achieves 30.07% throughput of the PIN approach at Zipfian $\theta = 0.99$.

The experiments of Octopus and XStore show that RDMA-based storage systems can gain significant performance enhancement from TeRM compared to the ODP and RPC approaches. TeRM saves physical memory and achieves comparable performance against the PIN approach.

7 Related Work

Extending local memory. With the advent of high-performance SSD and network, a host of works focus on extending local memory with the SSD or remote memory in recent years. They extend local memory from different levels, the application programming level [33,35], the virtual address level [11,18,21,28–30,42], and the hardware level [9,13,31]. Then the application process can run on a memory space larger than the physical memory and swap memory pages to the SSD or remote memory.

TeRM differs from these works in target problems and applications. These works focus on extending the private virtual memory of a CPU process and optimizing CPU page faults. Local memory is not exposed and only accessible by the process. Therefore, they target applications like in-memory graph processing systems (e.g., PowerGraph [17]) and big data systems (e.g., Spark [43]). TeRM extends the RDMA-attached memory exposed by the RNIC and tackles RNIC page faults. The memory is shared in the cluster and can be *concurrently* accessed by the server (via CPU) and multiple clients (via the RNIC). TeRM mainly aims at RDMA-based storage systems, e.g., Octopus [25] and XStore [37] in our evaluation.

ODP MR and RNIC page fault. Lesokhin et al. introduce ODP MR and page fault support for the RNIC [22], so that initializing an MR need not pin pages in physical memory. PART [32] also builds a mechanism to handle RNIC page faults on a prototype hardware platform. These works handle the exception in the hardware and thus are restricted by the limited hardware resources. TeRM proposes onloading exception handling from hardware to software.

Onloading from RNIC. Researchers from system and network communities also propose onloading functionalities from the RNIC to the CPU. For example, FaSST [20] and eRPC [19] reimplement reliability on the CPU, to address the scalability issues of the RC connection. Flor [24] onloads flow control from the RNIC to the CPU to support heterogeneous RNIC deployment. In contrast, TeRM targets page fault.

8 Conclusion

We present TeRM in this paper, an efficient approach to extending RDMA-attached memory with SSD. It onloads exception handling (i.e., RNIC page fault) from hardware to software. The experimental results on the microbenchmark and unmodified RDMA-based storage systems demonstrate the effectiveness of TeRM.

Acknowledgements

We sincerely thank our shepherd Joo-young Hwang for helping us improve the paper. We also thank the anonymous reviewers for their feedback. This work is supported by the National Key R&D Program of China (Grant No. 2021YFB0300500), the National Natural Science Foundation of China (Grant No. U22B2023 & 61832011), and the China Postdoctoral Science Foundation (Grant No. 2022M721828).

References

[1] Direct IO and page cache. https://lists.kernelnewbies.org/pipermail/kernelnewbies/2013-July/008660.html.

[2] Heterogeneous Memory Management (HMM) - Linux Kernel Documentation. https://www.kernel.org/doc/html/v5.19/vm/hmm.html.

[3] Intel Optane SSD P4800X. https://www.intel.com/content/www/us/en/products/sku/97161/intel-optane-ssd-dc-p4800x-series-375gb-2-5in-pcie-x4-3d-xpoint/specifications.html.

[4] Intel Optane SSD P5800X. https://www.intel.com/content/www/us/en/products/sku/201861/intel-optane-ssd-dc-p5800x-series-400gb-2-5in-pcie-x4-3d-xpoint/specifications.html.

[5] The page cache and page writeback. http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch15.html.

[6] Samsung PM9A3 SSD. https://semiconductor.samsung.com/ssd/datacenter-ssd/pm9a3/.

[7] Storage Performance Development Kit. https://spdk.io/.

[8] A parallel page cache: IOPS and caching for multicore systems. In *4th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 12)*, Boston, MA, June 2012. USENIX Association.

[9] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 971–985, 2019.

[10] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 599–616. USENIX Association, November 2020.

[11] 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 Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[12] 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.

[13] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2B-SSD: the case for dual, byte-and block-addressable solid-state drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 425–438. IEEE, 2018.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, page 401–414, USA, 2014. USENIX Association.

[15] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, New York, NY, USA, 2015. Association for Computing Machinery.

[16] Takuya Fukuoka, Shigeyuki Sato, and Kenjiro Taura. Pitfalls of infiniband with on-demand paging. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 265–275, 2021.

[17] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel computation on natural graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, pages 17–30, 2012.

[18] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

[19] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, February 2019. USENIX Association.

[20] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 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)*, pages 185–201, Savannah, GA, November 2016. USENIX Association.

[21] Gyusun Lee, Wenjing Jin, Wonsuk Song, Jeonghun Gong, Jonghyun Bae, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. A case for hardware-based demand paging. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 1103–1116, 2020.

[22] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. 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, page 449–466, New York, NY, USA, 2017. Association for Computing Machinery.

[23] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, pages 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Qiang Li, Yixiao Gao, Xiaoliang Wang, Haonan Qiu, Yanfang Le, Derui Liu, Qiao Xiang, Fei Feng, Peng Zhang, Bo Li, Jianbo Dong, Lingbo Tang, Hongqiang Harry Liu, Shaozong Liu, Weijie Li, Rui Miao, Yaohui Wu, Zhiwu Wu, Chao Han, Lei Yan, Zheng Cao, Zhongjie Wu, Chen Tian, Guihai Chen, Dennis Cai, Jinbo Wu, Jiaji Zhu, Jiesheng Wu, and Jiwu Shu. Flor: An open high performance RDMA framework over heterogeneous RNICs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI*

23), pages 931–948, Boston, MA, July 2023. USENIX Association.

[25] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, July 2017. USENIX Association.

[26] 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 13)*, pages 103–114, San Jose, CA, June 2013. USENIX Association.

[27] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. Balancing CPU and network in the cell distributed B-Tree store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 451–464, Denver, CO, June 2016. USENIX Association.

[28] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. Memory-mapped I/O on steroids. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 277–293, 2021.

[29] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. An efficient memory-mapped key-value store for flash storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 490–502, 2018.

[30] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped I/O for fast storage devices. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pages 813–827, 2020.

[31] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H Peter Hofstee. Thymesisflow: A software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 868–880. IEEE, 2020.

[32] Antonis Psistakis, Nikos Chrysos, Fabien Chaix, Marios Asiminakis, Michalis Giannioudis, Pantelis Xirouchakis, Vassilis Papaefstathiou, and Manolis Katevenis. Part: Pinning avoidance in rdma technologies. In *2020 14th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8, 2020.

[33] 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.

[34] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 433–448, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Suhas Jayaram Subramanya, Harsha Vardhan Simhadri, Srajan Garg, Anil Kag, and Venkatesh Balasubramanian. BLAS-on-flash: An efficient alternative for large scale ML training and inference? In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 469–484, 2019.

[36] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. CoRM: Compactable Remote Memory over RDMA. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1811–1824, New York, NY, USA, 2021. Association for Computing Machinery.

[37] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based ordered Key-Value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 117–135. USENIX Association, November 2020.

[38] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 233–251, Carlsbad, CA, October 2018. USENIX Association.

[39] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. Tmo: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 609–621, New York, NY, USA, 2022. Association for Computing Machinery.

[40] Bin Yan, Youyou Lu, Qing Wang, Minhui Xie, and Jiwu Shu. Patronus: High-Performance and Protective Remote Memory. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 315–330, Santa Clara, CA, February 2023. USENIX Association.

[41] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 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)*, pages 221–234, Boston, MA, February 2019. USENIX Association.

[42] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. Dilos: Do not trade compatibility for performance in memory disaggregation. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 266–282, New York, NY, USA, 2023. Association for Computing Machinery.

[43] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

A Artifact Appendix

Abstract

The artifact provides implementation source code and evaluation scripts of TeRM. It overloads the APIs of `libibverbs` and can be integrated with an existing RDMA application transparently by `LD_PRELOAD`.

Scope

The artifact helps understanding our design and implementation details better, including those that we do not mention in the paper due to space limit. It allows to reproduce the experimental results in the paper. It also provides examples for developers to integrate TeRM with their RDMA applications.

Contents

The implementation source code in the artifact contains two parts, the userspace shared library `libterm` (tLib) and the modified RNIC driver. Moreover, the artifact provides evaluation scripts and the third-party applications, including XStore and Octopus.

Hosting

The artifact is available at https://github.com/thustorage/TeRM. The `main` branch has the latest contents.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

Combining Buffered I/O and Direct I/O in Distributed File Systems

Yingjin Qian
Data Direct Networks

Marc-André Vef
Johannes Gutenberg University Mainz

Patrick Farrell
Whamcloud Inc.

Andreas Dilger
Whamcloud Inc.

Xi Li
Data Direct Networks

Shuichi Ihara
Data Direct Networks

Yinjin Fu
Sun Yat-Sen University

Wei Xue
Tsinghua University & Qinghai University

André Brinkmann
Johannes Gutenberg University Mainz

Abstract

Direct I/O allows I/O requests to bypass the Linux page cache and was introduced over 20 years ago as an alternative to the default buffered I/O mode. However, high-performance computing (HPC) applications still mostly rely on buffered I/O, even if direct I/O could perform better in a given situation. This is because users tend to use the I/O mode they are most familiar with. Moreover, with complex distributed file systems and applications, it is often unclear which I/O mode to use.

In this paper, we show under which conditions both I/O modes are beneficial and present a new transparent approach that dynamically switches to each I/O mode within the file system. Its decision is based not only on the I/O size but also on file lock contention and memory constraints. We exemplary implemented our design into the Lustre client and server and extended it with additional features, e.g., delayed allocation. Under various conditions and real-world workloads, our approach achieved up to 3× higher throughput than the original Lustre and outperformed other distributed file systems that include varying degrees of direct I/O support by up to 13×.

1 Introduction

High-performance computing (HPC) clusters traditionally store data on parallel file systems [4, 9, 14, 15, 49, 57]. They export local file or object storage from a collection of server nodes to clients, allowing applications on a client to access files on remote servers as if they were stored locally. Existing applications constantly scale to higher core counts and proportionally increase their I/O volume. New HPC applications from machine learning and AI are creating new access patterns that challenge previous optimizations for parallel file systems by increasing random accesses and heavy metadata traffic. As a result, I/O is increasingly becoming a performance bottleneck for many scientific applications.

File systems typically cache data and metadata in main memory to reduce the number of required I/Os to the storage backend. For example, Linux's default I/O mode is *buffered I/O* where the kernel caches read and write operations in the Linux page cache to help optimize I/O submitted to storage. Almost all standard applications running on a single server can benefit from page caching during I/O. Buffered I/O also improves the performance of many HPC applications that run on large clusters and store data on parallel file systems.

An alternative to buffered I/O is *direct I/O*. Files opened with the `O_DIRECT` flag bypass the caching layer in the kernel and send I/Os directly to the storage system. This is particularly useful when an application itself buffers read and write operations, avoiding a "double buffer" situation, such as in databases. To use direct I/O, an application must meet certain alignment criteria. The alignment constraints are usually determined by the disk driver, the disk controller, and the system memory management hardware and software. This requirement severely limits the use of direct I/O by applications.

Intuitively, buffered I/O should perform better than direct I/O because the read-ahead and write-back optimizations of buffered I/O can bring the performance of buffered I/O mode close to the level of memory access. Using direct I/O therefore typically results in a prolonged process. However, this paper shows that this is not always the case.

The reason is that caching data in the kernel page cache is not free, especially if the cached data has poor reuse characteristics. First, buffered I/O induces additional copy operations to move data between the kernel cache and the application. Second, the overhead of interacting with the kernel page cache and page management is considerable. Moreover, when memory becomes scarce, page reclamation must free old pages to allocate memory for the current I/O operation. The resulting cache thrashing can then significantly degrade performance.

An additional cost of buffered I/O in parallel file systems is the cost of managing complex distributed range locks to support client-side caching with strong consistency. If the file system locks only the necessary (small) portions of a concurrently accessed file, it requires many *remote procedure calls* (RPCs) between clients and the lock manager, while using larger expanding locks may lead to false lock contention between clients and many lock revocation messages [39].

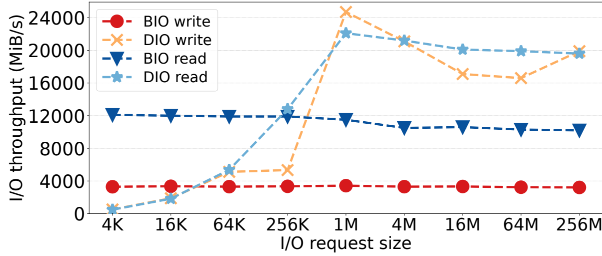


Figure 1: Local `ldiskfs` performance with various I/O sizes for buffered I/O (BIO) and direct I/O (DIO).

The primary benefit of direct I/O is to reduce CPU utilization for file reads and writes by eliminating the copy from the cache to the user buffer and minimizing the number of lock revocations. Therefore, the advantages and drawbacks of the two I/O modes complement each other. Buffered I/O simplifies programming and can yield performance benefits in many situations. However, for sequential I/O to very large files, direct I/O with large transfer sizes can provide the same or better performance as buffered I/O with much less CPU overhead and memory usage. In addition, direct I/O can also improve performance for small writes when many nodes with interleaved file offsets concurrently modify a file.

Figure 1 shows an example of this tradeoff when 16 threads run the `fio` benchmark on a local Lustre `ldiskfs` device. Each thread used separate files and wrote and read 20 GiB of data for I/O sizes between 4 KiB and 256 MiB. The write aggregation and read-ahead optimizations of buffered I/O resulted in a stable write performance of 3 GiB/s and a read performance of 11 GiB/s, almost independent of access sizes. This *performance wall* for buffered I/O depends on available memory for caching and page cache overhead and is independent of available storage bandwidth or number of the attached storage system. We also experienced the same behavior for other file systems like BeeGFS [25] or NFS [22].

The performance of direct I/O for small I/O sizes in Figure 1 is significantly lower than in buffered I/O mode. In this case, direct I/O suffers from latencies induced by synchronous writes to the storage backend. For bigger I/O sizes however, direct I/O benefits from not performing unnecessary copy operations and not having to manage the page cache, reaching a performance that exploits the potential of the backend SSDs.

To the best of our knowledge, we are the first to evaluate combining buffered I/O and direct I/O in the parallel file system itself. Based on our empirical results, we designed and implemented a new I/O path engine for the Lustre parallel file system that can automatically switch between buffered I/O and direct I/O modes. In contrast to previous work on BeeGFS [5], this switch is not only based on the size of requests but also considers memory pressure on compute nodes and lock contention on files. We also introduce a mechanism that supports adaptive switching between buffered I/O and direct I/O on storage servers.

We compare this new architecture with BeeGFS and OrangeFS [2]. We chose these two file systems as BeeGFS can switch between buffered I/O and direct I/O based on a fixed threshold [5], whereas OrangeFS in the tested implementation only performs direct I/O [17] on the client side. Our evaluation uses a variety of workloads, including microbenchmarks, macrobenchmarks, and real-world HPC workloads.

We show that our approach can effectively combine buffered I/O and direct I/O, selecting the best-performing I/O mode for a given I/O size and system state. Compared with the original Lustre version, our approach achieved up to 3× higher throughput for real-world workloads (that use many heterogeneous I/O sizes) and outperformed BeeGFS and OrangeFS by up to 13× and up to 10×, respectively. Moreover, we present the I/O statistics in which the I/O mode switch was triggered.

The remainder of this paper is organized as follows. First, Section 2 discusses the necessary background and motivates our work. Section 3 presents our new Lustre I/O engine. Next, Section 4 evaluates different parameter settings and compares our approach with BeeGFS and OrangeFS. Section 5 discusses related work, and we conclude with Section 6.

2 Background and motivation

In this section, we present a detailed comparative analysis between buffered I/O and direct I/O and then introduce the performance impact of page caching and I/O lock contention on buffered I/O to motivate our design for higher I/O performance in HPC systems.

2.1 Buffered I/O vs. direct I/O

Linux and most other operating systems offer buffered and direct I/O modes for file access. In the buffered I/O mode, the virtual file system first buffers all read and write requests in the kernel page cache. This is the default file access mode and is easy to integrate into applications as they do not need to deal with I/O size and alignment constraints. A major advantage of buffered I/O is that it can hide the latency of storage accesses when data is accessed more than once. The direct I/O mode in contrast transfers data directly between the application and the storage device without a data copy, but the data must meet specific size and alignment constraints.

Table 1 provides a high-level comparison of the two I/O modes. A key advantage of buffered I/O is its ability to prefetch data using read-ahead and to aggregate small writes using write-back caching. In both cases, small I/O requests from the application can be transformed into large I/O operations to the underlying storage system. Asynchronous write-back caching and read-ahead are perfect for hiding the latency of slow storage devices, such as spinning disks, and can perform close to the speed of the `memcpy()` operation. Buffered

Table 1: Comparison between two I/O modes.

I/O case	Buffered I/O	Direct I/O
Small I/O size	✓	X
High latency storage	✓	X
Unaligned I/O	✓	X
Large, sequential I/O	X	✓
Many running processes/nodes	X	✓
System under memory pressure	X	✓

I/O also has no requirements on the read and write size and alignment and is therefore convenient to use by programmers.

A major drawback of buffered I/O is its poor single-stream performance when data cannot be reused multiple times. It also creates many lock conflicts when multiple processes from different nodes write to a shared file on a networked file system with strong consistency guarantees. There is also contention within a single node's page cache when multiple cores are writing to a single file.

Direct I/O does not use page caching and transfers data directly between application memory and the storage device. It can provide near-device performance for large I/O sizes and does not slow down when scaling the number of processes and nodes. It can also reduce memory pressure because data is not prefetched or cached. The downside is that direct I/O cannot hide the latency for slow devices or small I/Os. Also, direct I/O requires that the I/O size and the offset in memory are aligned with the page size, so most applications must be significantly modified to use this I/O mode.

2.2 Impact of page caching and data copies on buffered I/O

The page cache used by buffered I/O induces additional copies between user space and the page cache. Its management requires, e.g., page allocation, locking, and LRU list management for aging and reclaiming. We therefore designed an experiment to measure the page cache overhead for sequential write operations from a single thread for a total I/O size of 2,560 GiB. We used the IOR benchmark [1] and the `perf` [28] profiling utility to collect and analyze the corresponding performance and trace data. Figure 2 shows the results for the local file system Ext4 and the network file systems Lustre and BeeGFS. These file systems spent about 20% of their time on copying data between the application and the page cache and more than 40% on page cache management. The figure also shows the resulting buffered I/O performance and the performance of direct I/O in the same setting.

This page cache overhead was also observed in previous work. For example, Corbet [16] describes that even (seemingly harmless) reads that span a dataset larger than the memory size can lead to page cache thrashing and huge performance drops once the page cache is full (see also [35, 44]). It is therefore interesting to understand why page cache management is so costly, even for such simple use cases.

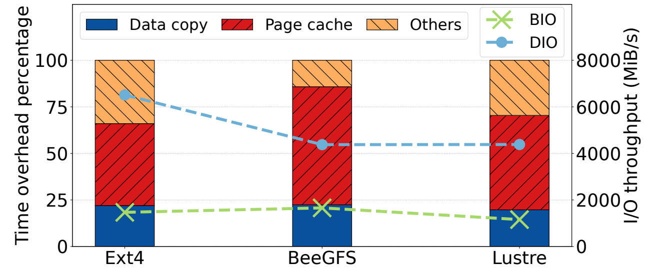


Figure 2: I/O time breakdown for buffered I/O writes.

Page cache management includes page allocation and page reclaim, among others. The page allocation process allocates clean pages for newly accessed data and adds the pages to the page cache. During memory pressure, Linux must reclaim some of the previously allocated pages from the cache to make room for newly allocated pages. In our example, reclaiming pages requires not only evicting pages from the page cache but also writing the data back to the storage system (see also [21]). A closer look at the profiling data shows that IOR running on BeeGFS, for example, spent more than 42% of its time in the `pagecache_get_page()` function. The reason is that `pagecache_get_page()` can only return under memory pressure after it has received a clean page. These clean pages must first be generated by the `kswapd` daemon. `kswapd` therefore constantly launched many `kworker` background threads that asynchronously wrote back these dirty pages using the native BeeGFS write functions, which in turn can impose an overhead and can lead to page cache thrashing. An additional page management overhead of 20% attributed to setting pages dirty by calling `__set_page_dirty_nobuffers()`.

Direct I/O does not interact with the page cache and can perform large sequential writes directly to the backend storage. The performance comparison in Figure 2 therefore shows that direct I/O on the local file system Ext4 can increase performance by nearly five times, while Lustre and BeeGFS have smaller gains because they require additional bulk data transfers over the network.

2.3 I/O locking and contention in Lustre

The Lustre file system stores data on object storage targets (OSTs) exporting local disk file systems through object storage servers (OSSs). Similarly, metadata is stored on metadata targets (MDTs) which are accessed through metadata servers (MDSs). Both data and metadata performance and capacity can be scaled by including more servers [9]. Lustre clients run on the compute nodes and access the storage and metadata servers through a high-speed network. Lustre supports client-side caching of data and metadata to reduce the impact of network round-trip times [45, 46]. It uses a *distributed lock*

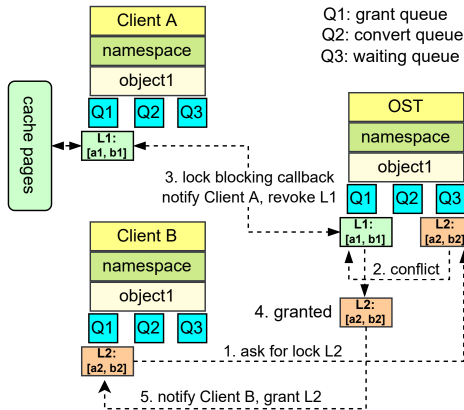


Figure 3: Lock blocking callback under write-back.

manager (DLM) [31, 37, 49] to protect the cached data and metadata from concurrent accesses by other clients. Lustre manages DLM locks in namespaces, where each Lustre OST or MDT has a separate lock namespace for its local objects.

Files are protected by read and write byte-range locks, allowing multiple clients to access or modify different parts of a shared file. A client requests a lock covering exactly the required I/O range (aligned with the page cache). The server attempts to optimize this request by expanding the lock to the largest non-conflicting range. The locks cached by a client are not released immediately and are instead revoked asynchronously through a callback in case of a lock conflict due to age or when exceeding cache sizes. Based on the locality principle, this can reduce lock traffic between clients and servers and improve performance, especially when only a single client accesses a file. Nevertheless, it can also result in heavy lock contention and false lock sharing for concurrent writes on a shared file from multiple clients [39].

Figure 3 shows the required steps to allocate a lock for write-back caching data in the range `[a2,b2]` by client B. We assume that client A previously wrote data to the same file and still keeps a lock `L1=<[a1, b1]>` in its local lock namespace. Now, client B requests a lock `L2=<[a2, b2]>` on the same object. The server detects that `L2` conflicts with `L1` because the two lock ranges `[a2, b2]` and `[a1, b1]` intersect and notifies client A to revoke `L1` via the lock-blocking callback. Client A then flushes the dirty pages to the server, clears the client cache, and releases `L1`. Afterward, the server grants `L2` to client B, and then client B can write to the file.

Lock conflict resolution and possible lock ping-pong are expensive processes and can significantly reduce I/O performance when writing to shared files. Therefore, we present a mechanism for using direct I/O with server-side locking to eliminate the lock callbacks for conflict I/Os next.

3 Design and implementation

Our design consists of four main components. Its switching algorithm automatically selects the I/O mode on both the client and the server to match request sizes and access patterns. Server-side adaptive locking reduces lock congestion on shared files when the I/O pattern has no access locality or when many clients access a file in parallel. Server-side delayed allocation improves strided I/O performance, and support for unaligned direct I/O accesses simplifies programmability. We have implemented our approach in the Lustre parallel file system. However, the general idea is applicable to other distributed file systems that also use a DLM.

3.1 Combining buffered I/O and direct I/O

We introduce a fully transparent hybrid I/O path engine that automatically switches between buffered I/O and direct I/O in the Lustre client and server.

***autoIO* – transparent direct I/O in the client:** *autoIO* uses the I/O request size, lock contention, memory pressure, and access locality to decide whether a client's I/O request should be handled as buffered or direct I/O. Algorithm 1 shows the corresponding decision tree for autoIO to automatically switch between buffered I/O and direct I/O.

If an I/O request is smaller than the *small I/O threshold*, autoIO uses buffered I/O. If the I/O request size is larger or equal to the *large I/O threshold*, autoIO uses direct I/O. Between both thresholds, autoIO uses buffered I/O by default and first checks whether the file is under lock contention due to conflicting accesses from multiple clients, and if so, switches to direct I/O. This is advantageous in combination with our adaptive server-side locking, which is discussed later in this section. Next, autoIO considers the client's current memory pressure and cache reuse. AutoIO therefore limits the number of cached pages of a file to 1 GiB (default) and switches to direct I/O when the cached pages or a corresponding cgroup reach 95% of their allowed limit. Also, if the I/O workload

Algorithm 1 The autoIO decision algorithm

```
1:  function DECIDE_IO_MODE(file, io_size, cfg)
2:      if (io_size ≥ cfg.large_io_threshold) then
3:          return DIO;
4:      else if (io_size < cfg.small_io_threshold) then
5:          return BIO;
6:      else if file_is_under_lock_contention(file) then
7:          return DIO;
8:      else if client_is_under_memory_pressure(file, cfg) then
9:          return DIO;
10:     else if file_lacks_access_locality(file, io_size, cfg) then
11:         return DIO;
12:     else
13:         return BIO;
```

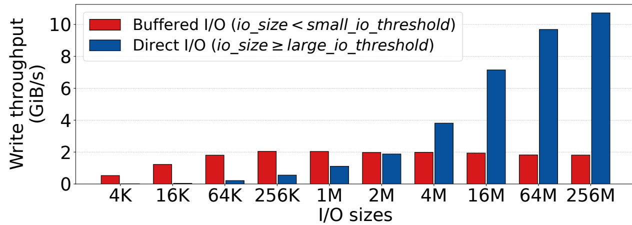


Figure 4: I/O streaming throughput for autoIO with various small and large I/O thresholds.

lacks access locality and the cached pages are not reused, autoIO switches to direct I/O for subsequent I/O accesses larger than the small I/O threshold.

For lock contention detection, autoIO leverages an already existing detection mechanism in Lustre. The other default parameters are based on preliminary experiments and are configurable by the user.

We ran I/O streaming experiments on a Lustre system with eight storage servers to determine the initial small and large I/O thresholds (see Section 4 for the experimental setup). We used IOR [1] with various I/O request sizes ranging from 4 KiB to 256 MiB. Figure 4 presents the results for each I/O size for two cases: First, the small I/O threshold was set larger than the I/O request size, resulting in buffered I/O. Second, the large I/O threshold was set smaller than the I/O request size, resulting in direct I/O. Based on the results, we set the large I/O threshold to 2 MiB by default. We further set the small I/O threshold to 32 KiB. Therefore, if Lustre detects lock contention, memory pressure, or a lack of access locality, autoIO switches to direct I/O in the range [32 KiB, 2 MiB). Section 4 provides additional I/O statistics when a specific case was triggered and shows the performance benefits of switching to direct I/O across workloads within this range.

Note that autoIO does not switch to buffered I/O when a user opens a file with the `O_DIRECT` flag.

Adaptive server-side locking for direct I/O Bulk-synchronous applications [10] represent some of the most dominant workloads in HPC, where applications typically alternate compute and I/O phases at step boundaries. Often, such workloads access a single shared file at a step boundary, where each process accesses its own non-conflicting region. Examples include Nek5000 [20] for computational fluid dynamics, VPIC for large-scale plasma physics simulations [8, 12], and checkpointing to a single shared file [7].

This workload type is so common that it is included in the IO500 [32] *ior-hard-write* benchmark where each MPI rank concurrently writes into a single shared file with an I/O size of 47,008 bytes in strided I/O mode. Although the I/O regions do not overlap, this can cause significant lock contention. The reason is that clients must obtain page-aligned extent locks before performing their non-page-aligned I/Os. This can result in many ping-pong lock callbacks that negatively affect I/O throughput.

For direct I/O, our implementation leverages an existing server-side locking mechanism in which clients send their I/O requests directly to the server, which acquires the DLM lock on the client's behalf. Compared to the client-side extent lock, the server-side extent lock has a much lower latency because the server can take the lock before proceeding with the bulk transfer (and free it directly after), saving lock round-trip traffic. Therefore, when a file is under lock contention, the benefits of direct I/O shift towards smaller I/O sizes.

For autoIO, this means that, according to our experiments, it is beneficial to use direct I/O already below the large I/O size threshold but above the small I/O size threshold (see Algorithm 1 line 6) when a file is under lock contention. To determine whether a lock resource is under contention and to what degree, our detection algorithm uses a sliding window counter [59]. If at least 16 (by default) conflicting lock requests are seen over a sliding window of 4 seconds (by default), the file object is considered under contention, which the server reports to the client via the reply. Overall, this can increase the efficiency for I/O requests that fall between the small and large I/O thresholds, as shown later in Section 4.

Server-side adaptive write-back and write-through Lustre servers implement a thread pool model for incoming requests from many clients in parallel. Specifically, Lustre uses an out-of-band data transfer mode, combined with *remote direct memory access* (RDMA) network transfers, to minimize CPU and memory utilization while providing high throughput and scalability. For large incoming I/O requests, each I/O service thread uses a pre-allocated buffer (between 4 MiB and the maximum bulk RPC size) for bulk data transfers to avoid the overhead of the kernel page cache. Depending on the server load, the number of I/O threads can vary from 2 to 512 threads. Therefore, the total memory requirement can be several GiB (RPC size times 512 threads). By default, all I/O requests are immediately submitted to storage by the I/O service thread to avoid memory contention (*write-through*).

Although this works well for large I/O requests, especially when using large bulk RPCs (up to 64 MiB in Lustre), the write-through mode does not fully utilize the available disk bandwidth for small I/O requests (see Figure 1). This is particularly noticeable for latency-sensitive I/O requests, such as when many small files are written and read.

We added an adaptive write-back cache mode to improve I/O performance on the server for such latency-sensitive use cases. Similar to how the Lustre client can switch to direct I/O for non-`O_DIRECT` requests, the Lustre server can switch to buffered I/O mode (*write-back*). According to the results in Figure 1, all I/O requests smaller than 64 KiB are processed in write-back mode using the page cache. For larger I/O requests, Lustre uses its default write-through mode via direct I/O. This allows the server to use the available memory to cache small

I/Os while not overwhelming the cache with large I/Os.

Moreover, we do not implement server-side read-ahead. The reason is that 1. the server memory is a limited resource shared by all clients, and read-ahead data may overwhelm the cache space on a server; and 2. client-driven read-ahead in Lustre is a more efficient way and has already achieved very good performance in most cases.

Note that when using write-back, limited data availability after a crash is a challenge, with the potential to lose non-persisted data. However, this is also the case with buffered I/O in general. Here, `fsync()` is essential to ensure data consistency and durability within the file system. This is in accordance with POSIX which states that a successful `write()` does not guarantee that the data has been committed to disk unless `fsync()` is called. Interestingly, this applies also to direct I/O. Although it bypasses the client's kernel cache, a storage system may or may not apply `O_DIRECT` to other layers of the I/O stack where caching is used. Therefore, it is important to use `fsync()` to flush all cached data to disk.

Overall, our work does not affect metadata durability and consistency as the file system can always recover to a consistent state after a crash. For example, Lustre's recovery mechanism can resend (replay) uncommitted I/O operations on clients to the server once the server resumes operations [43].

Cross-file batching for buffered writes For single files, Lustre clients accumulate an application's dirty pages and asynchronously send them as large bulk RPCs (1 MiB) to the storage servers. This method avoids many small RPCs, and it is therefore more network and disk-efficient. For many small files, however, there may not be enough dirty pages to accommodate a full bulk RPC, delaying the write-back operation. Moreover, I/O RPCs sending dirty pages to the storage servers are restricted to a single file and thus many small RPCs are sent.

Cross-file batching for buffered writes is an optimization strategy for such use cases that involve many small writes across many small files. It batches dirty pages of multiple files into one large bulk RPC, improving network efficiency. The configurable threshold `small_write_threshold` (default 64 KiB) allows Lustre to distinguish whether a file is small enough to benefit from this optimization. Essentially, Lustre clients maintain a small-file list that contains files below the threshold. Once enough dirty pages are placed in the list, batched I/O bulk RPCs are sent to the storage servers.

Consistency challenges Whenever our algorithm triggers transparent and dynamic switching between buffered I/O and direct I/O, there is the potential for data regions from the two I/O modes to overlap. For instance, on the Lustre client, a file region could be written via direct I/O while parts have not been flushed yet and remain in page cache as they were part of a prior buffered I/O operation. If not handled properly, this can cause a consistency conflict. A similar situation could occur on the server with the above-described write-back cache and the default direct I/O path.

On the client, overlapping regions are detected, and dirty pages in that region are flushed first before direct I/O is performed. On the server side, with write-back enabled, dirty pages are not flushed and are reused instead as part of the direct I/O operation. Thus, because clients must flush their cache and servers merge overlapping data, our approach does not change Lustre's strong consistency guarantees. Similar arguments hold for aligning unaligned direct I/O, which we discuss next.

3.2 Unaligned direct I/O

One of the challenges of using direct I/O is that the `O_DIRECT` open flag typically imposes alignment constraints on the length and address of user space buffers and the file offset of I/Os, where the I/O size and offset must be a multiple of 512 bytes and the memory buffer address must also be aligned to 512 bytes [26]. In this context, the 512-byte boundary refers to the logical block size of the underlying storage device, although modern devices use a sector size of 4096 bytes or more [36]. If not all conditions are true, direct I/O is not supported and the error code `EINVAL` is returned.

In general, I/O alignment provides several benefits. For example, it can resolve conflicting *read-modify-write* (RMW) operations on the same block. However, not all applications can align their I/O. This is especially true for applications with a complex I/O stack that is not under the control of the application. Therefore, to maximize the benefits of direct I/O, the underlying file system should handle misaligned I/O in the kernel. We implemented a buffering scheme in the Lustre client to address this challenge. When the user buffer is misaligned, Lustre creates an aligned buffer in the kernel by remotely reading data outside the user buffer up to the alignment boundary. Direct I/O then uses this aligned buffer.

Nevertheless, because aligning a user buffer potentially requires additional memory allocation and data copying, it may be less efficient than using an existing user buffer. However, our analysis showed that allocating and copying to an aligned buffer in the kernel still outperforms buffered I/O, especially for large I/O sizes. This is because buffered I/O spends less than 20% of its time allocating a buffer and copying data to that buffer, while the remaining time is spent locking page caches and managing the kernel cache.

3.3 Efficient RAID I/O via delayed allocation

When Lustre receives I/O write requests for a small file, blocks are allocated at write time, even if the data is written in write-back mode. This strategy can lead to severe file fragmentation for strided I/O (discussed above) when multiple clients are writing data to a single file and are therefore constantly allocating blocks simultaneously. Since magnetic disks are still

used as the primary backend for storing data in parallel file systems, we try to reduce file fragmentation by delaying block allocation on the backend storage.

A common technique to mitigate such file fragmentation is to use *delayed (block) allocation* by deferring data block allocation until the last possible moment before data is flushed to disk. Delayed block allocation is featured in many modern file systems, e.g., EXT4 [13], XFS [29], or BtrFS [48] to reduce fragmentation [51].

Therefore, we have enabled delayed allocation in write-back mode on the server to collect and merge small or non-contiguous I/O requests into large, contiguous I/O requests. This can reduce head thrashing on magnetic disks. It can also reduce the number of RMWs on RAID systems by consolidating full stripes before flushing the data. Small fragmented I/O, on the other hand, immediately writes the data to the RAID controller cache before flushing it in an RMW fashion, significantly impacting I/O performance.

Delayed allocation uses the kernel's write-back mechanism to flush dirty pages and allocate blocks at flush time. By default, Linux's periodic flushing interval is five seconds, during which the disk bandwidth may be underutilized. To flush data continuously, we leverage *extent status trees* (available in Ext4 and ldiskfs) – a data structure to track the status of delayed allocation extents. When a server receives a write request from a client, it allocates an in-memory delayed extent and inserts it into the delayed extent status tree. During insertion, the server looks for other delayed extents to form a contiguous extent. If Lustre detects that a merged extent can form a full extent write, e.g., offset and length are both 1 MiB aligned, the dirty pages from this extent are flushed by a worker thread. In summary, this allows larger continuous I/O buffers to be flushed to the underlying RAID disk system outside of the periodic flushing interval, reducing RMW operations caused by otherwise small extents and improving the overall I/O performance in the process.

4 Evaluation

This section evaluates the performance and benefits of our work under various workloads, including microbenchmarks, macrobenchmarks, and real application workloads. Most of these experiments are compared to BeeGFS and OrangeFS. We chose these two distributed file systems for comparison because BeeGFS supports switching between buffered I/O and direct I/O based on a fixed threshold value [5], whereas OrangeFS, in the tested implementation, only performs direct I/O on the client side [17].

This section uses abbreviations for various I/O and file system modes. `BIO` (buffered I/O) and `DIO` (direct I/O) refers to the I/O mode across several benchmarks, i.e., whether `O_DIRECT` was used with `open()`. File system modes refer to configuration changes applied to the three file systems used.

First, our new Lustre features can be toggled and configured independently, allowing us to investigate the impact of each. The following abbreviations for Lustre are used: 1. `vanilla` for the original Lustre version; 2. `autoIO` for the client-side decision algorithm; 3. `svrWB` for server-side write-back caching; 4. `delalloc` for the delayed (block) allocation; and 5. `XBatch` for cross-file batching of buffered writes.

For BeeGFS, we used two file cache modes on the clients [5]: 1. `buffered` mode representing the *default* file cache mode for write-back and read-ahead by using several static buffers; and 2. `native` mode that relies on the Linux page cache. BeeGFS's `native` mode offers the `tuneFileCacheBufSize` parameter (512 KiB by default) for switching to direct I/O above the threshold. In that case, all I/O operations bypass the page cache, communicating directly with the storage servers. `Native` is therefore comparable to our `autoIO`, which further considers additional parameters.

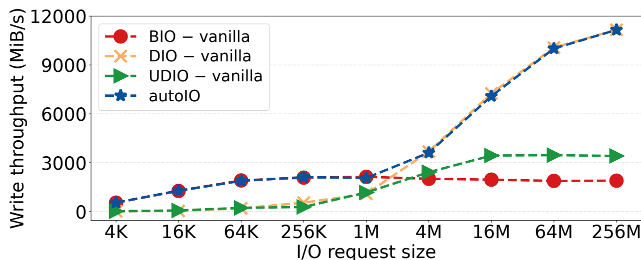
For OrangeFS, we used two server-side I/O modes: 1. `alt-aio` mode which is the *default* for accessing data on the storage servers by using buffered I/O via asynchronous I/O; and 2. `directio` mode that uses direct I/O to access data on the storage backend. Similar to our `svrWB` caching, which allows Lustre also to use buffered I/O on the servers, OrangeFS servers can therefore operate in buffered and direct I/O mode.

Our experiments were run on a Lustre cluster consisting of 4 MDTs, 8 OSTs, and 32 client nodes. The servers used a DDN AI400X2 Appliance backend (20 $\times$ SAMSUNG 3.84 TiB NVMe, 4 $\times$ IB-HDR100 100 Gbps), running Lustre version 2.15.58. All clients used an Intel Gold 5218 processor, 96 GiB of DDR4 memory, and ran CentOS 8.7 Linux. All nodes were interconnected using InfiniBand IB-HDR100.

BeeGFS used a storage architecture similar to Lustre, with the same hardware and configurations. Both clients and servers were running BeeGFS 7.4.0. For OrangeFS, we used version 2.10.0 running CentOS 8.7 Linux on both servers and clients. Each metadata storage target was configured with two metadata server instances, and each data storage target was configured with one data server instance. Thus, there are eight metadata instances and eight data server instances in total. The client kernel module for OrangeFS with CentOS 8.7 does not integrate with the Linux page cache, and all client I/Os (both direct and buffered I/O modes) are performed synchronously. Unless otherwise specified, a given file is striped across eight storage targets by default, and the stripe size is 1 MiB for all file systems.

4.1 Microbenchmarks

This section presents the experiments and results for various microbenchmark workloads using IOR and mdtest [1], which have become popular benchmarking tools in HPC [23].



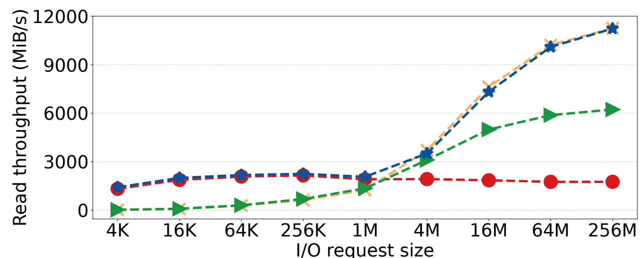


Figure 5: Single I/O stream throughput for Lustre.

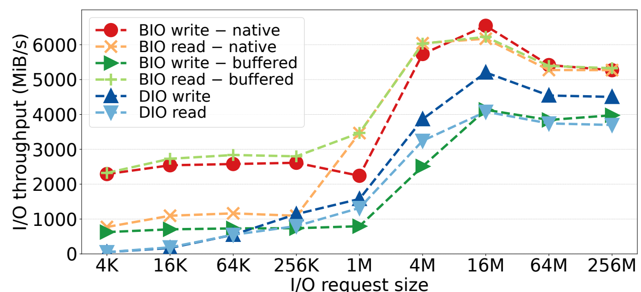


Figure 6: Single I/O stream throughput for BeeGFS.

Single I/O stream throughput First, we present the single I/O stream throughput for BIO, DIO, and unaligned DIO (UDIO) in vanilla Lustre and with our autoIO compared with BeeGFS and OrangeFS. In this case, IOR ran a single process, writing and reading data 2× the memory size with I/O sizes varying from 4 KiB to 256 MiB on the client.

Figure 5 visualizes the results that are overall similar to the local ldiskfs's I/O throughput earlier in this paper (see Figure 1). The I/O sizes for unaligned direct I/O were increased by 8 bytes in this experiment as we aimed to observe whether aligning unaligned direct I/O can yield benefits over buffered I/O. As expected, the performance of unaligned direct I/O was lower than aligned direct I/O due to the overhead of extra memory allocation and copying. However, since it did not need to interact with page caching and management, it could outperform buffered I/O when the I/O size was larger than 4 MiB. Our autoIO mode took advantage of the two I/O modes and avoided their shortcomings, achieving the best overall performance over the entire range of I/O sizes.

Figure 6 shows the results for this workload running BeeGFS. BeeGFS achieved its highest throughput in native mode. Overall, BeeGFS achieved at most 6.5 GiB/s and

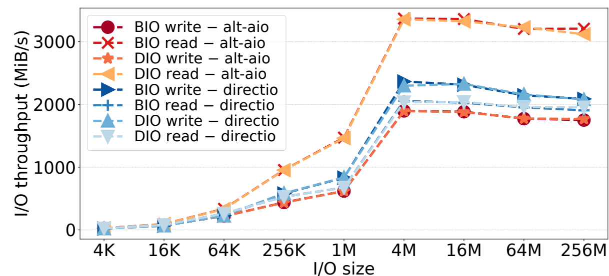


Figure 7: Single I/O stream throughput for OrangeFS.

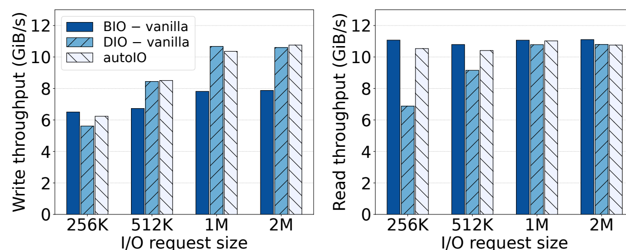


Figure 8: Lustre's I/O throughput for 16 processes.

6.2 GiB/s for writes and reads, respectively. In comparison, Lustre's autoIO achieved 11.1 GiB/s (1.7×) and 11.2 GiB/s (1.8×) for writes and reads, respectively.

OrangeFS reached the lowest I/O throughput of the three file systems (see Figure 7) with at most 2.3 GiB/s and 3.3 GiB/s for the respective writes and reads as all client I/Os were executed synchronously. Thus, the performance of direct I/O and buffered I/O are nearly the same. The results also indicate that the write performance using the server-side direct I/O mode (directio) has a slight edge over the alt-aio mode (buffered I/O). Read performance, on the other hand, achieved about a 50% higher throughput with alt-aio compared to direct I/O mode as it re-used data in the page cache.

Multiple I/O stream throughput Next, we ran IOR with 16 processes on a single client, sequentially writing and reading 80 GiB in file-per-process mode for I/O sizes ranging from 256 KiB to 2 MiB. Each file used only a single stripe object. The goal of this experiment was to investigate the trade-off phase of autoIO within the [256 KiB, 1 MiB] range and whether autoIO selects the best-performing I/O mode.

Figure 8 presents the results. Note that the I/O sizes in the range [256 KiB, 1 MiB] are larger than the small I/O threshold but smaller than the large I/O threshold. In this range, the efficiency of direct I/O and the performance optimizations due to write-back and read-ahead prefetching for buffered I/O are in a trade-off phase (see Algorithm 1), with autoIO almost reaching the best of both modes.

Table 2 lists the I/O statistics for the 512 KiB I/O size, that is, when autoIO switched to direct I/O due to memory

Table 2: I/O statistics for parallel I/O with 16 processes on 1 client node for a 512 KiB I/O size.

I/O Type	DIO (memory pressure)	DIO (cache overuse)	BIO (default)
Write	0	2,457,520	163,920
Read	45,593	5,993	4,738

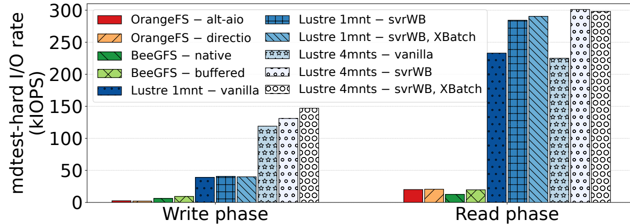


Figure 9: IO500's *mdtest-hard* performance for ten nodes.

pressure or cache over-usage. `Default` represents autoIO staying in buffered I/O mode. For instance, it shows that most writes were switched to direct I/O because more data was cached than allowed (see Section 3 in the page caches. For reads in autoIO, 45,593 I/Os were performed in direct I/O due to the memory pressure. In summary, these I/O statistics demonstrate that autoIO considers memory pressure and access locality to avoid excessive caching to obtain similar performances among the two I/O modes.

IO500's mdtest-hard workload To evaluate the I/O improvements for many small files, we ran mdtest in the *mdtest-hard* configuration of the IO500 "10-node challenge" benchmark. *mdtest-hard* generates many small files (with a size of 3091 bytes) in a single directory. We used 10 clients with 16 processes each, creating, writing, and reading 128,000 files per rank. Figure 9 shows the *mdtest-hard-write* and *mdtest-hard-read* results.

For Lustre, we compared several configurations: First, we used either one (`1 mnt`) or four separate Lustre mount points (`4 mnts`). For the latter, each mount point was assigned four MPI ranks. This technique mitigates locking congestion in the *virtual file system* (VFS) during parallel file creation in a shared directory. We also focused on measuring the impact of the `XBatch` and `svrWB` optimizations. Due to the many small files in this workload, autoIO and delayed allocation did not improve performance and are omitted. Finally, we used Lustre's *Data on MDT* (DoM) feature, which improves small file performance by placing small files only on the MDT and eliminating additional RPCs to the OSTs.

We also ran these experiments with BeeGFS (`native`, `buffered`) and OrangeFS (`alt-aio`, `directio`), albeit only working on a single mount point. In this configuration and compared to the vanilla Lustre case (`4 mnts`), they reached at most 2% of the performance for writes with OrangeFS

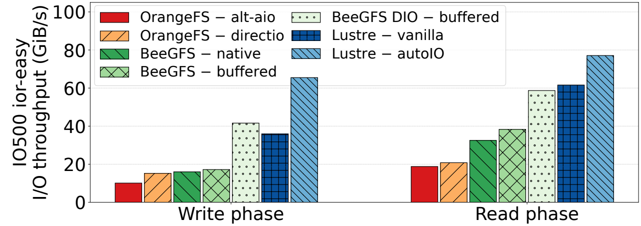


Figure 10: IO500's *IOR-easy* performance for ten nodes.

(`alt-aio`, `directio`) and 8% with BeeGFS (`buffered`). For reads, OrangeFS (`alt-aio`, `directio`) and BeeGFS (`buffered`) reached at most 9% of Lustre's performance.

For Lustre, we measured a performance benefit when our optimizations were enabled. In the write case (`4 mnts`) and with `svrWB` caching, the server sent a reply to the client as soon as the data was copied to the page cache, resulting in a 10% performance improvement over vanilla Lustre (`4 mnts`). With XBatch added, we achieved a 20% increase in performance. In the read case, the client's MPI ranks are offset so that the readers cannot benefit from the client's page cache. However, with `svrWB` caching enabled, the data is still available in the server's page cache, resulting in a 33% improvement over vanilla Lustre. For the `1 mnt` cases, the create-write performance did not benefit from the optimizations due to VFS locking congestion.

IO500's IOR-easy workload In this section, we evaluate the performance of all three file systems using IO500's *IOR-easy* use case with 10 nodes and 16 processes per node. Each process wrote and read in 16 MiB I/O size requests to a dedicated file for at least 300 seconds. Further, each file is striped across eight storage targets in all cases.

Figure 10 illustrates that `Lustre - autoIO` outperformed BeeGFS and OrangeFS in all configurations. This is because Lustre used direct I/O on both clients and servers with such a large I/O size (as opposed to `Lustre - vanilla` which used buffered I/O). BeeGFS used the less efficient buffered I/O on the servers for this I/O size, even when using direct I/O on the clients, resulting in inferior performance.

IO500's IOR-hard workload While *IOR-easy* represents a common sequential workload that generally works well for distributed file systems (and especially autoIO), *IOR-hard* creates a cyclic data distribution with an I/O size of 47,008 bytes that is neither aligned to page or file system block boundaries. Moreover, all processes access a single shared file with a segment count of 40,000. Figure 11 presents the results for the *IOR-hard* workload.

Generally, this is a challenging workload, particularly when aligning unaligned direct I/O and because of lock traffic overheads. In the case of Lustre and unaligned buffered writes, the client must first lock the object and read the unaligned page(s)

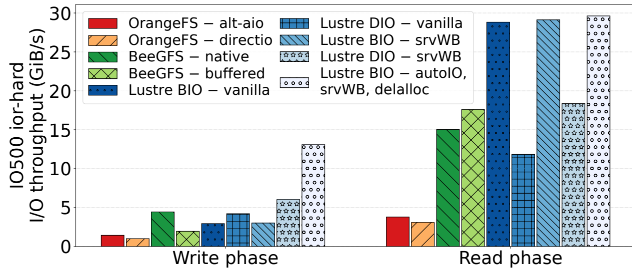


Figure 11: IO500's *IOR-hard* performance for ten nodes.

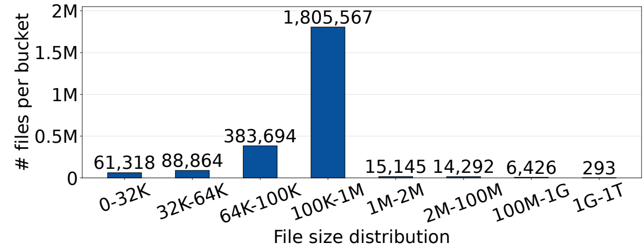


Figure 12: File size distribution of a large dataset.

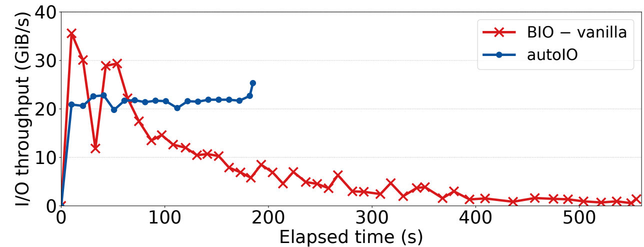


Figure 13: *dcp* I/O bandwidth over time for Lustre.

within the I/O range. Only then can the modified parts of the page(s) be updated. These unaligned RMW operations on the pages can severely affect performance. Although buffered writes can be aggregated in the page cache on a client, the writes must be contiguous on a single client and must be protected by the DLM extent lock. As the requested DLM extent lock must be page-aligned on the client, it may conflict with lock requests from other similarly page-aligned clients. This results in unnecessary lock contention when obtaining the DLM extent lock from the server.

Our I/O statistics revealed that even though buffered I/O accessed smaller file fragments, it still generated 3.5 million lock callbacks (35 per I/O segment). This is in contrast to unaligned direct I/O, which generated no lock callbacks due to server-side locking. As a result, the I/O throughput increased from 3 GiB/s to 4.2 GiB/s. By using `autoIO` as well as enabling `svrWB` (see Section 3.1) and `delalloc` (see Section 3.3), the write bandwidth reached 13 GiB/s.

Note that with server-side delayed allocation, data only needs to be written to the page cache without block allocation, thus reducing the I/O request latency. Moreover, we measured a reduction in file allocation fragments from 100K to about 35K, leading to a significant increase in 1 MiB size writes (100K+) that match our stripe size. Further, we enabled Lustre's overstriping feature [19, 38] which can improve I/O performance by allowing multiple stripes per OST. In this case, we set the `lfs setstripe` parameter to use 1,000 stripe objects, i.e., 125 stripes per OST. Overall, our additions improved performance by 4× compared with vanilla Lustre.

In the case of *ior-hard-read* and unaligned direct I/O reads, the I/O throughput increased from 11 GiB/s to 18 GiB/s with only `svrWB` caching enabled. This is due to many page cache read hits on the server that avoid reading from disk. The results also show that buffered I/O achieved much better read performance due to client-side read-ahead, reaching 30 GiB/s. With `autoIO`, reads achieved similar performances since the DLM lock for read operations from multiple clients were compatible. Therefore, the reads were performed in buffered I/O mode from the start.

Both BeeGFS and OrangeFS do not support unaligned direct I/O and used buffered I/O mode in this case. BeeGFS's native mode was slightly faster than Lustre with normal direct I/O. Lustre, on the other hand, with our `autoIO` and `delalloc`, was 2.9× faster than BeeGFS (`native`) for writes and 1.7× faster than BeeGFS (`buffered`) for reads.

4.2 mpiFileUtils/dcp workload

This section examines the I/O performance over time when copying a large file using *mpiFileUtils/dcp* [50]. Further, we copied an 8.8 TiB heterogeneous dataset, containing millions of files in a directory hierarchy with more than 10K directories (see the file size distribution in Figure 12), and compared the bandwidth with the three file systems in different modes. Dcp segments a large file into fixed-size chunks (4 MiB by default) and places a new distributed work item for each chunk in a global queue. The data copies are then distributed across multiple MPI ranks.

We first ran dcp on 32 nodes (16 processes each), writing and reading a 4 TiB to and from a single file (on the same file system) in parallel with a chunk size of 4 MiB. The source and target file were both striped across 8 OSTs. Figure 13 shows the bandwidth variation over time. Due to the large chunks that triggered direct I/O in the autoIO algorithm, it achieved a 3× performance improvement at 20 GiB/s and a more stable throughput than the buffered I/O performance in the vanilla case. As expected, the buffered I/O performance dropped drastically once most memory was consumed by the page cache. The observed lock callback count was over 77K. The memory pressure, due to interacting with the kernel's page cache, and false lock callbacks all contributed to the performance drop.

Next, we copied the entire dataset using dcp across 10 nodes (16 processes each) and a chunk size of 4 MiB. In this

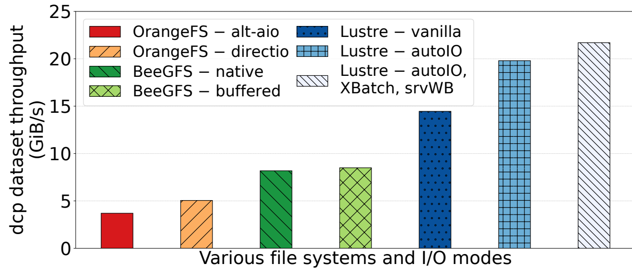
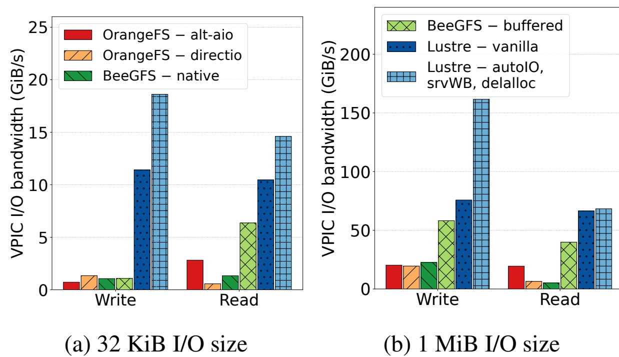


Figure 14: I/O throughput for copying a dataset with *dcp*.



(a) 32 KiB I/O size (b) 1 MiB I/O size

Figure 15: VPIC-IO bandwidth for various file systems.

case, we set autoIO's large I/O threshold to 512 KiB, which is similar to BeeGFS's default `native` mode.

Figure 14 presents the performance for the three file systems with various configurations. Overall, Lustre with `autoIO`, `XBatch`, and `svrWB` reached 21.7 GiB/s and outperformed vanilla Lustre by 1.5×, BeeGFS (`buffered`) by ∼2.6×, and OrangeFS (`directio`) by 4.3×. These results match the above *IOR-easy* conclusions in file system capabilities and demonstrate that our additions benefit both large files and heterogeneous datasets.

4.3 VPIC-IO workload

VPIC-IO is a macrobenchmark that represents the I/O kernel [12] of a large-scale plasma physics simulation that can compute, e.g., the reconnection and turbulence in solar weather. We ran the VPIC-IO kernel via h5bench [33] which uses an emulated compute time of two seconds for each time step and writes random particle data. Specifically, each MPI rank writes many particles into a single shared HDF5 file for a certain number of time steps, called *particle dump*.

We ran VPIC-IO on 32 nodes (16 processes each) for 8,192 and 262,144 particles and I/O sizes of 32 KiB and 1 MiB, respectively. In contiguous storage mode, the HDF5 metadata header is separate from the dataset data, with the data itself stored in one contiguous block in the HDF5 file. This led to MPI-IO starting not from 0 but at a specific offset (`mpi_off=2104` in our case), which is equal to the size of

Table 3: I/O statistics for VPIC-IO and 32 KiB I/O size.

Count	DIO (large I/O)	DIO (lock contention)	BIO (small I/O)	BIO (default)
AutoIO	0	807,876	1,043	11,324

the metadata header. Since the offset of MPI-IO is not page-aligned, all I/O operations were unaligned. Figure 15 shows the write and read bandwidth for OrangeFS, BeeGFS, and Lustre. Lustre performed best among the three file systems. With `autoIO` and a 1 MiB I/O size, for instance, the write performance reached 3× of Lustre's `vanilla` performance as most I/O operations are switched to direct I/O due to lock contention.

The I/O statistics for the 32 KiB I/O size workload for Lustre and autoIO are listed in Table 3. Since the I/O size was small, I/O was initially handled in buffered I/O mode. Because I/O locks must be page-aligned in Lustre, this resulted in significant lock contention on the server, mainly due to unaligned I/O. This caused the clients to switch 807,876 I/O requests to direct I/O. 1,043 requests were handled in buffered I/O due to the small I/O size, and 11,324 were processed with buffered I/O by default. Finally, we monitored that more than 50% of the 32 KiB I/O requests were merged into a 1 MiB full stripe using `svrWB` caching and `delalloc`.

4.4 Nek5000 turbulent pipe flow workload

The Nek5000 application [20] for computational fluid dynamics (CFD) is a bulk-synchronous application. Its workflows can define step boundaries when Nek5000 should flush vector data, vector statistics, or write checkpoints. At each step, all ranks participate in writing to a single shared file at predefined offsets, depending on the number of participating processes.

In our experiments, we ran a turbulent pipe flow workload [47] on 32 nodes with 16 processes each. Contrary to VPIC-IO, this workload represents a real application and includes the computational component as well. In this experiment, Nek5000 executed 1,000 time steps (running for about 10 minutes), writing the corresponding vectors at each step, accounting for 600 GiB of data in total. Further, we used the Darshan profiling tool [40] to collect all I/O access sizes of

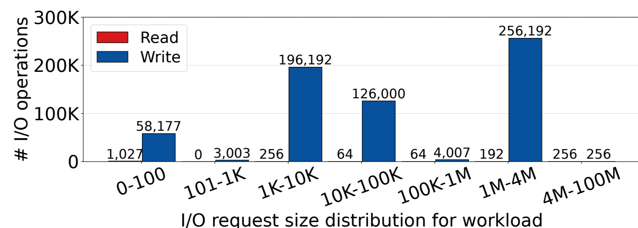


Figure 16: Nek5000's turbPipe workload I/O access sizes.

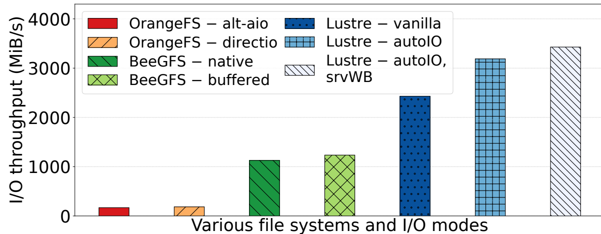


Figure 17: Nek5000 bandwidth for various file systems.

Table 4: I/O statistics for Nek5000's turbPipe workload.

Count	DIO (large I/O)	DIO (lock contention)	BIO (small I/O)	BIO (default)
AutoIO	128,000	132,000	372,281	65

this workload (see Figure 16), revealing a broad profile of small and large I/O requests.

Figure 17 presents the I/O bandwidth of Nek5000. Lustre with `autoIO` and `svrWB` caching reached 3,428 MiB/s and outperformed `vanilla` Lustre by 1.4×, BeeGFS (`buffered`) by ∼2.8×, and OrangeFS (`directio`) by 18.5×. Table 4 shows the I/O statistics using `autoIO`. The I/O throughput improved by more than 10% with `svrWB` caching enabled due to many small I/O requests. Note that the I/O sizes are proportional to the number of participating processes. Thus, with half the number of nodes and processes, e.g., 16 nodes, the I/O sizes become larger, increasing the effectiveness of direct I/O. In this case, Lustre with `autoIO` achieved 60% higher bandwidth than `vanilla` Lustre.

5 Related work

This section discusses the related work concerning avoiding the page cache, locking file system resources for strided I/O, dynamic I/O path, and I/O mode selection.

Direct access and page cache avoidance When direct I/O is used to bypass the kernel's page cache, it considerably impacts performance if used incorrectly while greatly benefiting large I/O operations [27]. Other approaches were made to directly access the storage device without involving the kernel to avoid overhead in the kernel's I/O stack [30,54]. Aerie [54], for example, processes metadata within a trusted metadata server in user space, potentially involving costly RPCs that can affect scalability.

For *non-volatile main memories* (NVMM) devices, page cache overhead due to data copies is unnecessary and can be directly addressed by the NVMM device. The DAX (direct access) feature in the Linux kernel uses the DAX interface to bypass the page cache and exposes the persistent memory driver [18, 52]. Nevertheless, Simurgh [41] showed that DAX is often insufficient to expose the NVMM device's native performance. Other Linux features, e.g., `POSIX_FADV_DONTNEED fadvise()`, can reduce the impact of page caching for reads to discard a data range from the page cache after reads, avoiding page reclaiming during page allocation. A similar idea was developed for the Linux kernel with the `RMF_UNCACHED` flag, where pages are only added to the page cache for the duration of a read and removed after [16]. However, this approach still suffers from page management overhead, especially when using high-speed networks and storage backends.

Our work focuses on storage backends used by distributed file systems where bypassing the page cache is not always the best option. Therefore, we showed the importance of dynamically deciding whether to bypass the page cache based on both the I/O access patterns and the system state.

Cache strategies Numerous caching strategies targeting distributed file systems were designed over the years, including client-side write-back caching [45], I/O caching middlewares [60], employing machine learning to automate caching [24], or leveraging existing node-local storage devices [46]. Our implementation does not replace or add to these existing strategies. Instead, autoIO identifies requests for which it is more beneficial to bypass the client's page cache. AutoIO thus relieves cache pressure and allows cache strategies to be optimized for the remaining cached data. The break-even point at which direct I/O becomes useful depends not only on the given I/O size but also on lock contention, memory pressure, and overall access locality. AutoIO is the first transparent client-side algorithm to handle these parameters as part of the parallel file system.

Distributed locking Lock managers are vital to modern distributed software, e.g., to provide strong consistency in a distributed file system. In general, concurrent applications require locking that allows coordinated access to shared resources. Chubby [11] provides a coarse-grained synchronization mechanism between servers for reliability and availability in a loosely coupled distributed system. Lustre, on the other hand, must protect and coordinate access to shared resources by many clients in parallel to guarantee that both data and metadata remain consistent [9]. It further offers applications a user space API to request locks in advance for specific data ranges that are not allowed to expand [39]. This allows applications to avoid false lock conflicts beforehand and works well in strided I/O access patterns. Nonetheless, modifying applications to fully control their I/O behavior is not always feasible, especially when using I/O libraries. Our server-side adaptive locking can transparently react to lock-congested files without changing the application.

Dynamic I/O path Applications have widely different workload statistics that can even suddenly change with new

application types entering the HPC space, e.g., in the cases of data-driven application and AI workloads [10]. To accommodate past and future applications, parallel file systems often adopt the "one-size-fits-all" solution, such as Lustre [9], GPFS [49], and BeeGFS [25]. However, depending on the application and storage system, this can reduce I/O performance. Dynamic I/O paths can therefore be valuable to fit more closely to an application's I/O behavior.

Xiuqiao et al. [34] use a file handle-rich scheme in PVFS [14] providing a framework to enable a dynamic, fine-granular, and client-side I/O path section at runtime on a per-job basis. The *balanced placement I/O* (BPIO) library [56] intelligently allocates I/O paths for a parallel file system, binding a client to a storage target while evenly distributing the I/O traffic across components to proactively avoid contention points. To reduce I/O contention in an HPC environment, TAPP-IO [42] provides dynamic shared data placement mitigating resource contention and load-imbalance to improve application I/O, while *iez* [55] offers a transparent and adaptive control plane for balanced data placement.

Rather than focusing on data placement, we dynamically select the most suitable I/O path for each I/O request by using already existing I/O protocols, i.e., direct I/O, which can be challenging for developers to use directly.

Rigid vs. dynamic I/O mode Buffered I/O is still used as the default I/O mode in most situations. However, direct I/O is employed by other distributed file systems as well, e.g., BeeGFS and OrangeFS. BeeGFS's native mode, for instance, can switch from buffered I/O to direct I/O for I/O requests that are larger than the tunable `tuneFileCacheBufSize` parameter (512 KiB by default). Yet, direct I/O triggered on the client does not affect server behavior, which still relies on buffered I/O. Conversely, OrangeFS offers the *alt-aio* (default) and direct I/O modes that only affect server I/O behavior. The former uses a thread-based implementation for asynchronous I/O using `pread()` and `pwrite()`. Similarly, an NFS [22] server can use synchronous or asynchronous I/O (decided before launch). However, the cache protocols in BeeGFS, OrangeFS, and NFS are all non-coherent.

In contrast to the above file systems and to the best of our knowledge, we are the first to implement and evaluate a fully adaptive and transparent dynamic I/O path on both the file system client and server, using the most suitable I/O path in a given situation. In addition, we take file lock contention, memory pressure, and page cache reuse statistics into account to decide whether to use buffered I/O or direct I/O (even if unaligned) with strong consistency.

6 Conclusion and future work

This paper has presented a new approach to transparently and dynamically switch between buffered I/O and direct I/O in distributed file systems. We have shown the benefits of both I/O modes over a range of I/O sizes and have presented a client-side I/O mode switching algorithm that considers not only I/O sizes but also file lock contention and memory constraints. Other features include an adaptive server-side write-back cache, alignment of unaligned I/O, delayed allocation, and I/O request batching. We have achieved these features without compromising Lustre's strong consistency guarantees. Overall, our experimental results over several microbenchmarks, marcobenchmarks, and real-world workloads have shown that our approach reached up to 3× higher throughput than original Lustre and outperformed other distributed file systems by up to 13×.

Our future work covers several directions. First, we plan to conduct an extensive analysis of the performance impact of diverse I/O sizes, thresholds, file system configurations, and application workloads to further optimize and specialize autoIO's behavior. Second, we aim to modify autoIO's decision thresholds further so that they can automatically adapt depending on the system state by adopting previous work, e.g., machine learning-based prediction to optimize I/O bandwidth [6,53,58]. Third, we will design a server-side algorithm to switch between write-back and write-through modes that further considers the server state.

Acknowledgments and availability

We sincerely thank our shepherd Jinkyu Jeong and the anonymous reviewers for helping us improve our paper significantly.

This research was supported by the National Key R&D Program of China under Grant No.2022YFB4500304, the Natural Science Foundation of China under Grant No. 61832020 and No. 62332021.

This work was partially funded by the European Union's Horizon 2020 and the German Ministry of Education and Research (BMBF) under the "Adaptive multi-tier intelligent data manager for Exascale (ADMIRE)" project; Grant Agreement number: 956748-ADMIRE-H2020-JTI-EuroHPC-2019-1.

This work was also partially supported by the BMBF under the "Federated Digital Infrastructures for Research on Universe and Matter" FIDIUM project; Grant Agreement number: 05P21UMRC1.

References

[1] Ior and mdtest. https://github.com/hpc/ior, 2022. Accessed on Sep, 19, 2023.

[2] Orangefs. http://www.orangefs.org/, 2023. Accessed on Sep, 19, 2023.

[3] FAST '24: "Combining Buffered I/O and Direct I/O in Distributed File Systems" - Artifacts Description. https://doi.org/10.5281/zenodo.10425915, 2024.

[4] Abutalib Aghayev, Sage A. Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP), Huntsville, ON, Canada, October 27-30*, pages 353–369, 2019.

[5] BeeGFS. Client side caching modes. https://doc.beegfs.io/latest/advanced_topics/client_caching.html. Accessed on Sep, 19, 2023.

[6] Babak Behzad, Surendra Byna, Prabhat, and Marc Snir. Optimizing i/o performance of hpc applications with autotuning. *ACM Transactions on Parallel Computing (TOPC)*, 5(4):1–27, 2019.

[7] John Bent, Garth A. Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC), November 14-20, Portland, Oregon, USA*, 2009.

[8] Kevin J. Bowers, B. J. Albright, Lilan Yin, B. Bergen, and Thomas J. T. Kwan. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas*, 15(5):055703, 03 2008.

[9] Peter Braam. The lustre storage architecture. *CoRR*, abs/1903.01955, 2005.

[10] André Brinkmann, Kathryn M. Mohror, Weikuan Yu, Philip H. Carns, Toni Cortes, Scott Klasky, Alberto Miranda, Franz-Josef Pfreundt, Robert B. Ross, and Marc-Andre Vef. Ad hoc file systems for high-performance computing. *J. Comput. Sci. Technol.*, 35(1):4–26, 2020.

[11] Michael Burrows. The chubby lock service for loosely-coupled distributed systems. In *7th Symposium on Operating Systems Design and Implementation (OSDI), November 6-8, Seattle, WA, USA*, pages 335–350, 2006.

[12] Suren Byna, Andrew Uselton, D Knaak Prabhat, and Yun He. Trillion particles, 120,000 cores, and 350 tbs: Lessons learned from a hero i/o run on hopper. In *Cray user group meeting*, 2013.

[13] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. Ext4: The next generation of ext2/3 filesystem. In *Linux Storage and Filesystem Workshop, February 12–13, 2007, San Jose, CA*, 2007.

[14] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *4th Annual Linux Showcase & Conference 2000, Atlanta, Georgia, USA, October 10-14*, 2000.

[15] Fahim Chowdhury, Yue Zhu, Todd Heer, Saul Paredes, Adam Moody, Robin Goldstone, Kathryn M. Mohror, and Weikuan Yu. I/O characterization and performance evaluation of beegfs for deep learning. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP), Kyoto, Japan, August 05-08*, pages 80:1–80:10, 2019.

[16] Jonathan Corbet. Buffered i/o without page-cache thrashing. https://lwn.net/Articles/806980/, 2019. Accessed on Sep, 19, 2023.

[17] OrangeFS Documentation. Orangefs configuration file. https://docs.orangefs.com/configuration/admin_ofs_configuration_file/. Accessed on Sep, 19, 2023.

[18] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (ATC), Santa Clara, CA, USA, July 12-14*, pages 719–731, 2017.

[19] Patrick Farrell. Overstriping: Extracting maximum shared file performance. https://wiki.lustre.org/images/b/b3/LUG2019-Lustre_Overstriping_Shared_Write_Performance-Farrell.pdf, 2019. Accessed on Sep, 19, 2023.

[20] Paul Fischer, James Lottes, and Henry Tufo. Nek5000. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2007.

[21] 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 (ATC), Boston, MA, USA, July 11-13*, pages 15–28, 2018.

[22] Thomas Haynes. Network file system (NFS) version 4 minor version 2 protocol. *RFC*, 7862, 2016.

[23] Michael Hennecke. Understanding daos storage performance scalability. In *Proceedings of the HPC Asia 2023 Workshops*, pages 1–14, 2023.

[24] Herodotos Herodotou. Autocache: Employing machine learning to automate caching in distributed file systems. In *35th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2019, Macao, China, April 8-12, 2019*, pages 133–139. IEEE, 2019.

[25] Frank Herold, Sven Breuner, and Jan Heichler. An introduction to beegfs. https://www.beegfs.io/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf, 2014. Accessed on Sep, 19, 2023.

[26] IBM. Considerations for the use of direct i/o (o_direct). https://www.ibm.com/docs/en/storage-scale/5.1.8?topic=applications-considerations-use-direct-io-o-direct, 2023. Accessed on Sep, 19, 2023.

[27] Russell Joyce and Neil C. Audsley. Exploring storage bottlenecks in linux-based embedded systems. *SIGBED Rev.*, 13(1):54–59, 2016.

[28] Linux Kernel. Linux kernel profiling with perf. https://perf.wiki.kernel.org/index.php/Tutorial. Accessed on Sep, 19, 2023.

[29] Dohyun Kim, Kwangwon Min, Joontaek Oh, and Youjip Won. Scalexfs: Getting scalability of XFS back on the ring. In *20th USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, February 22-24*, pages 329–344, 2022.

[30] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. Nvmedirect: A user-space I/O framework for application-specific optimization on nvme ssds. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage), Denver, CO, USA, June 20-21*, 2016.

[31] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. Vaxclusters: A closely-coupled distributed system. *ACM Trans. Comput. Syst.*, 4(2):130–146, 1986.

[32] Julian M. Kunkel, John Bent, Jay Lofstead, and George S. Markomanolis. White paper: Establishing the io-500 benchmark. Technical report, The Virtual Institute for I/O, 2017.

[33] Tonglin Li, Suren Byna, Quincey Koziol, Houjun Tang, Jean Luca Bez, and Qiao Kang. h5bench: Hdf5 i/o kernel suite for exercising hpc i/o patterns. In *Proceedings of Cray User Group Meeting, CUG*, volume 2021, 2021.

[34] Xiuqiao Li, Limin Xiao, Meikang Qiu, Bin Dong, and Li Ruan. Enabling dynamic file I/O path selection at runtime for parallel file system. *The Journal of Supercomputing*, 68(2):996–1021, 2014.

[35] 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 (ATC), July 15-17*, pages 897–910, 2020.

[36] Linux man-pages 6.04. open(2) — linux manual page. https://man7.org/linux/man-pages/man2/open.2.html, 2023. Accessed on Sep, 19, 2023.

[37] Ajay Mohindra and Murthy V. Devarakonda. Distributed token management in calypso file system. In *Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing (SPDP), Dallas, Texas, USA, October 26-29*, pages 290–297, 1994.

[38] Michael Moore. Exploring lustre overstriping for shared file performance on disk and flash. 2019.

[39] Michael Moore, Patrick Farrell, and Bob Cernohous. Lustre lockahead: Early experience and performance using optimized locking. *Concurrency and Computation: Practice and Experience*, 30(1), 2018.

[40] Nafiseh Moti, André Brinkmann, Marc-André Vef, Philippe Deniel, Jesús Carretero, Philip H. Carns, Jean-Thomas Acquaviva, and Reza Salkhordeh. The I/O trace initiative: Building a collaborative I/O archive to advance HPC. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W 2023, Denver, CO, USA, November 12-17, 2023*, pages 1216–1222. ACM, 2023.

[41] Nafiseh Moti, Frederic Schimmelpfennig, Reza Salkhordeh, David Klopp, Toni Cortes, Ulrich Rückert, and André Brinkmann. Simurgh: a fully decentralized and secure NVMM user space file system. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, page 46. ACM, 2021.

[42] Sarah Neuwirth, Feiyi Wang, Sarp Oral, and Ulrich Brüning. Automatic and transparent resource contention mitigation for improving large-scale parallel file system performance. In *23rd IEEE International Conference on Parallel and Distributed Systems (ICPADS), Shenzhen, China, December 15-17*, pages 604–613, 2017.

[43] Sarp Oral, Feiyi Wang, David Dillow, Galen M Shipman, Ross G Miller, and Oleg Drokin. Efficient object storage journaling in a distributed parallel file system. In *FAST*, volume 10, pages 1–12, 2010.

[44] Yoshihiro Oyama, Shun Ishiguro, Jun Murakami, Shin Sasaki, Ryo Matsumiya, and Osamu Tatebe. Reduction of operating system jitter caused by page reclaim. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS), Munich, Germany, June 10*, pages 9:1–9:8, 2014.

[45] Yingjin Qian, Wen Cheng, Lingfang Zeng, Marc-André Vef, Oleg Drokin, Andreas Dilger, Shuichi Ihara, Wusheng Zhang, Yang Wang, and André Brinkmann. Metawbc: Posix-compliant metadata write-back caching for distributed file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Dallas, TX, USA, November 13-18*, pages 56:1–56:20, 2022.

[46] Yingjin Qian, Xi Li, Shuichi Ihara, Andreas Dilger, Carlos Thomaz, Shilong Wang, Wen Cheng, Chunyan Li, Lingfang Zeng, Fang Wang, Dan Feng, Tim Süß, and André Brinkmann. LPCC: hierarchical persistent client caching for lustre. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Denver, Colorado, USA, November 17-19*, pages 88:1–88:14, 2019.

[47] Saleh Rezaeiravesh, Ricardo Vinuesa, and Philipp Schlatter. A statistics toolbox for turbulent pipe flow in nek5000. Technical report, KTH, Fluid Mechanics and Engineering Acoustics, 2019.

[48] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[49] Frank B. Schmuck and Roger L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST), January 28-30, Monterey, California, USA*, pages 231–244, 2002.

[50] Danielle Sikich, Giuseppe Di Natale, Matthew LeGendre, and Adam Moody. mpifileutils: A parallel and distributed toolset for managing large datasets. Technical report, Lawrence Livermore National Lab (LLNL), Livermore, CA, USA, 2017.

[51] Chenlei Tang, Jiguang Wan, Yifeng Zhu, Zhiyuan Liu, Peng Xu, Fei Wu, and Changsheng Xie. Rafs: A raid-aware file system to reduce the parity update overhead for ssd raid. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1373–1378. IEEE, 2019.

[52] Nick-Andian Tehrany. *Evaluating Performance Characteristics of the PMDK Persistent Memory Software Stack*. PhD thesis, Vrije Universiteit Amsterdam, 2020.

[53] Abdul Jabbar Saeed Tipu. Hpc io and seismic data performance optimization using anns prediction based auto-tuning. 2023.

[54] Haris Volos, Sanketh Nalli, Sankaralingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: flexible file-system interfaces to storage-class memory. In *Ninth Eurosys Conference (EuroSys), Amsterdam, The Netherlands, April 13-16*, pages 14:1–14:14, 2014.

[55] Bharti Wadhwa, Arnab Kumar Paul, Sarah Neuwirth, Feiyi Wang, Sarp Oral, Ali Raza Butt, Jon Bernard, and Kirk W. Cameron. iez: Resource contention aware load balancing for large-scale parallel file systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Rio de Janeiro, Brazil, May 20-24*, pages 610–620, 2019.

[56] Feiyi Wang, Sarp Oral, Saurabh Gupta, Devesh Tiwari, and Sudharshan S. Vazhkudai. Improving large-scale storage system performance via topology-aware and balanced data placement. In *20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), Hsinchu, Taiwan, December 16-19*, pages 656–663, 2014.

[57] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *7th Symposium on Operating Systems Design and Implementation (OSDI), November 6-8, Seattle, WA, USA*, pages 307–320, 2006.

[58] Li Xu, Thomas Lux, Tyler Chang, Bo Li, Yili Hong, Layne Watson, Ali Butt, Danfeng Yao, and Kirk Cameron. Prediction of high-performance computing input/output variability and its application to optimization for system configurations. *Quality Engineering*, 33(2):318–334, 2021.

[59] yongjoon. Rate limiter — sliding window counter. https://medium.com/@avocadi/rate-limiter-sliding-window-counter-7ec08dbe21d6, 2022. Accessed on Sep, 19, 2023.

[60] Dongfang Zhao, Kan Qiao, and Ioan Raicu. Hycache+: Towards scalable high-performance caching middleware for parallel file systems. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2014, Chicago, IL, USA, May 26-29, 2014*, pages 267–276. IEEE Computer Society, 2014.

A Artifact Appendix

Abstract

The *Artifacts Description* (AD) of this paper [3] provides detailed documentation of the used configurations for all file systems and experiments.

Scope

The AD has been made *available*. It includes detailed reference instructions to set up, deploy, and configure each used file system and each presented experiment. Please note that these artifacts are not functional due to the complexity of fully configuring and testing a Lustre installation automatically. However, the AD makes all instructions available for reference, making it possible to run similar experiments in similar configurations as provided in the paper.

Contents

The AD is sectioned into three main parts: Prerequisite information, e.g., software dependencies, installing the three used file systems, i.e., Lustre with autoIO, BeeGFS, and OrangeFS, and the description of all experimental workloads.

Prerequisites We describe the experimental setup and the required software dependencies for CentOS 8.4. Further, we provide requirements for setting up the cluster environment with the corresponding environment variables.

File system installation For Lustre, we provide detailed documentation of installing and configuring a Lustre parallel file system from scratch. We present information on Linux InfiniBand drivers (OFED) and provide reference Linux Bash scripts for installing and deploying the Lustre clients and servers (including autoIO). Moreover, we link the corresponding Git branches and pull requests to Whamcloud's issue and project tracking software (*JIRA*). These include all code changes from this paper and further serve as a reference for the status of each feature (eight in total).

Moreover, we provide the reference Bash scripts for installing the BeeGFS and OrangeFS clients and servers from scratch.

Experimental workloads We present the reference Bash scripts for each experiment and figure used in the paper. This includes setting file system configurations, e.g., enabling our server-side write-back, installing and running the evaluated applications. We further include a dedicated document for each experiment type in addition to the corresponding scripts.

Hosting

The AD is hosted on Zenodo and GitHub. GitHub supplements the long-term Zenodo repository and offers easy access to the artifacts' documentation and scripts. The corresponding versions are mirrored between Github and Zenodo, i.e., v1 on Zenodo mirrors v1.0 on GitHub. Please note that a DOI [3] via Zenodo is available for referencing these artifacts.

OmniCache: Collaborative Caching for Near-storage Accelerators

Jian Zhang (Rutgers University), Yujie Ren (Rutgers University), Marie Nguyen (Samsung),
Changwoo Min (Igalia), Sudarsun Kannan (Rutgers University)

Abstract

We propose *OmniCache,* a novel caching design for near-storage accelerators that combines near-storage and host memory capabilities to accelerate I/O and data processing. First, OmniCache introduces a *"near-cache" approach*, maximizing data access to the nearest cache for I/O and processing operations. Second, OmniCache presents *collaborative caching* for concurrent I/O and data processing by using host and device caches. Third, OmniCache incorporates a *dynamic model-driven offloading* support, which actively monitors hardware and software metrics for efficient processing across host and device processors. Finally, OmniCache explores the *extensibility for newly-introduced CXL*, a memory expansion technology. OmniCache demonstrates significant performance gains of up to 3.24X for I/O workloads and 3.06X for data processing workloads.

1 Introduction

The growth in data volume and demand for high-performance data processing is driving innovative storage architectures. Traditional approaches with centralized processing and frequent data movement face performance limitations and high costs [9, 10, 44]. To address this, vendors have introduced near-storage data processing devices, bringing processing capabilities closer to storage [11, 15, 28]. These architectures leverage accelerators and host processors to enhance processing power and potentially reduce data movement and associated overheads. Realizing these benefits requires effective management and utilization of these resources. State-of-the-art near-storage designs have explored various approaches to accelerate I/O and data processing. These include using storage as a raw block device [33], developing near-storage key-value stores [13, 17, 24, 34], and creating near-storage file systems [9, 21, 31]. Additionally, application-customized techniques have been proposed [37, 39].

Utilizing memory buffers on near-storage accelerators is crucial for mitigating the impact of higher storage latency and limited bandwidth. Near-storage memory offers advantages such as localization and high bandwidth, making it an advantageous buffering medium near computational units. However, near-storage memory capacity is typically smaller than traditional host-level RAM, as demonstrated by prior studies and commercial products [11, 15]. Therefore, effective techniques that collaboratively use device and host-level memory and processors become crucial, minimizing data movement between storage and host layers, resulting in accelerated data processing and regular I/O operations (e.g., read, write).

While state-of-the-art near-storage designs improve I/O or data processing performance, they either lack any memory caching support [9, 31, 33] or fail to exploit device-level memory in collaboration with host-level memory for caching [17, 24, 44]. The absence of caching support or the failure to exploit near-storage memory for I/O and data processing increases storage access and data movement between the host and the device (e.g., fetching a 4KB block for a 1KB application request). Similarly, the absence of a collaborative host and device memory caching causes applications to stall due to cache eviction delays. Finally, prior designs use simplistic metrics to offload data processing (e.g., computing power) without considering storage-centric metrics (e.g., data distribution, I/O-to-processing ratio, data movement bandwidth, and queuing costs), leading to suboptimal performance.

To tackle the challenges above, we propose ***OmniCache***, a cross-layered system software design that exploits the combined capabilities of near-storage accelerators, host CPUs, and their memory (DRAM) resources to accelerate I/O and data processing. At its heart, OmniCache introduces *a novel principle, "near-cache",* which focuses on maximizing data access on the closest cache, effectively combining the strengths of the host (such as higher memory capacity and more CPUs) and the device (being nearer to the storage) while mitigating their limitations. OmniCache employs a horizontal paradigm where application threads can concurrently store and access data from the host and the device caches, thereby improving the aggregate bandwidth and data access latency. Toward designing OmniCache, we make the following contributions:

Near-cache I/O: Firstly, we optimize I/O performance using a near-cache mechanism that simultaneously utilizes host-

level cache (*HostCache*) and device cache (*DevCache*). This near-cache approach maximizes cache utilization for various I/O access patterns, transferring only the data sizes requested by an application thread instead of the entire block from storage to the host.

Collaborative Caching for Concurrent I/O: Unlike hierarchical caching approaches where threads must wait for cache eviction to complete when a cache (e.g., HostCache) is full, OmniCache's horizontal paradigm allows threads to update the other cache (e.g., DevCache) until the eviction is complete and reduces application stalls. To locate data stored in these caches or on the disk, we introduce a scalable, host-managed indexing mechanism known as *OmniIndex*. OmniIndex utilizes a per-file interval tree equipped with a fine-grained range lock, enabling threads to access both the host and device caches concurrently for non-conflicting blocks [7].

Collaborative Processing with Dynamic Offloading: Second, we develop a dynamic offloading mechanism driven by an offloading model to accelerate data processing by leveraging HostCache and DevCache collaboratively. The mechanism enables concurrent data processing across the host and the devices and uses the caches to buffer the intermediate processing state. Beyond simple processing (e.g., data checksums and compression), we develop support for complex processing operations (e.g., K-nearest neighbor search).

Exploiting CXL.mem Capabilities: Finally, to demonstrate the adaptability of OmniCache beyond conventional NVMe-based near-storage, we exploit byte-addressable Compute Express Links (CXL) [1] with memory expansion capabilities (*CXL.mem*) to coordinate between host and device caches, reduce data movement costs and queuing delays.

End-to-end Evaluation: We evaluate OmniCache with microbenchmarks and real-world applications, including LevelDB [3] and K-Nearest Neighbor search [32]. OmniCache's near-cache I/O principle, collaborative use of DevCache and HostCache for concurrent I/O and to dynamically offload processing functions provide significant performance gains. Compared to the state-of-the-art near-storage file systems without caching [9] and those with host-only caching [44], OmniCache achieves 3.24x and 1.52x performance gains, respectively. Application write stalls are reduced by up to 2×. The collaborative approach to concurrently process data across the host and the device provides up to 3.06x gains over state-of-the-art FusionFS [9]. Finally, LevelDB [3] and data-intensive KNN [32] show up to 5.15x gains, highlighting the practical benefits of OmniCache.

2 Background and Motivation

We first present the background and related work on near-storage data processing and caching, followed by their limitations and analysis that motivates OmniCache design.

2.1 Background and Related Work

We now review prior near-storage processing studies in terms of (1) hardware trends, (2) software advancements, (3) near-storage file systems and OS support for data processing, and finally, (4) in-memory caching for storage.

Hardware Near-storage Processing Trends: Despite advancements in SSD and NVM technologies, data access and movement overheads remain dominant in I/O stacks. To address these overheads, hardware manufacturers are enhancing storage-level compute resources in near-storage processing devices like Computational Storage Devices (CSD). These devices are equipped with powerful ARM or RISC-V cores [21, 31, 34, 36, 38], FPGAs [13, 33], and significant DRAM capacity. Recent developments include CSDs with 16GB device RAM and 16-core Cortex processors [15]. They offer predefined functions and customization options to eliminate data movement between the host and the device while improving performance and flexibility [16, 38]. Moreover, CSDs such as ScaleFlux [35] and Newport [15] seamlessly integrate processor, memory, and SSD control. This integration eliminates off-chip communication and enables fast data transfer to device compute units, presenting a novel opportunity to explore the utilization of device resources.

Additionally, the emerging CXL technology (Compute Express Link) holds promise for hardware-supported memory expansion across accelerators and remote hosts [1, 19, 26]. CXL encompasses protocols such as CXL.io, CXL.cache, and CXL.mem, offering device types with different data-coherence guarantees. It enables host CPUs to expand and access device memory, with the potential to cache data on device memory for accelerating I/O and data processing.

Software Support for Near-storage Acceleration: To fully exploit the potential of near-storage accelerators, considerable software advancements have been explored to minimize data movement costs, which accelerate and efficiently leverage near-storage accelerators. Table 1 summarizes the capabilities and limitations of existing systems. Designs such as INSIDER [33] offload compute tasks to FPGA-based CSD using a block-based interface. Key-value interface designs, such as POLARDB [13], PINK [17], and KEVIN [24] offload database-specific computation to near-storage. Furthermore, NearPM [37] and SmartRec [39] focus on customized application-level optimizations or system-level guarantees.

In contrast to these systems, near-storage file system designs offload the file system closer to the storage while maintaining a POSIX-like interface. Systems such as DevFS [21] and CrossFS [31] adopt this approach by offloading metadata structures to improve performance and efficiency. FusionFS [9] compared in this work, combines file system operations with computation steps and incorporates device-level task scheduler and durability and recoverability mechanisms.

Data Processing Support: Various near-storage data processing systems have been explored. POLARDB [13] develops new application logic to accelerate applications by offloading data to FPGA-based key-value stores. λ-IO [44] utilizes an OS file system as a unified IO stack to manage computation and storage resources across the host and device.

Properties	Block-based	KV-based	Host FS	Dev-FS	Dev-FS with caching
System	Insider [33]	KV-SSD [34] PINK [17] KEVIN [24]	λ-I/O [44]	CrossFS [31] FusionFS [9]	**OmniCache**
Direct-I/O	✗	✗	✗	✓	✓
Use host Cache	✓	✓	✓	✗	✓
Use device Cache	✗	✗	✗	✗	✓
Concurrent host and device I/O processing	✗	✗	✗	✗	✓
Dynamic Offload	✗	✗	Partial	✗	✓
CXL support	✗	✗	✗	✗	✓

Table 1: Capabilities and Limitations of State-of-the-art Near-storage approaches. The last column shows our proposed OmniCache.

It extends eBPF for executing functions on heterogeneous hardware and provides additional programming interfaces for customized computational logic. Finally, FusionFS introduces $CISC_{Ops}$ abstraction that combines I/O and data processing operations to reduce application changes and overheads associated with I/O operations, such as system calls, data movement, communication, and other software overheads [9].

In-memory Caching for Storage: Caching I/O data in DRAM is critical for modern I/O stacks [12]. Traditional file systems rely on an OS-managed page cache, which can introduce user-to-kernel boundary crossings and substantial software overheads, often nullifying the benefits of fast storage devices [10, 21, 27, 31, 33]. While several file systems for devices like PM have disabled caching [41, 43], others are exploring user-level caching support [2, 23, 27, 30, 45]. However, *none of these storage designs have explored or designed system software for collaboratively managing on-device and host memory buffers for accelerating I/O or data processing*.

Host-level Caching for Near-Storage Processing: Caching designs like λ-IO [44] exploit host OS-level caches and only offload processing for data not in the host OS. While useful, these designs fail to consider or exploit device-level caches and suffer from the following challenges: Firstly, they lack direct I/O support, incur system call overhead, and must trap into the OS, all adding significant overheads. Secondly, they incur high data movement between the device and the host. Thirdly, host-level caching designs like λ-IO fail to concurrently support I/O and data processing across the host and the device. Fourthly, these designs lack CXL support. In contrast, OmniCache provides direct I/O bypassing the OS, reduces data movement between the device and the host, provides concurrent I/O and processing capability across the host and the device, and support for *CXL.mem* and *CXL.io*. Finally, while both OmniCache and λ-IO provide model-driven offloading mechanisms, λ-IO overlooks critical factors like device cache and command queue delays, significantly impacting performance (demonstrated in §6).

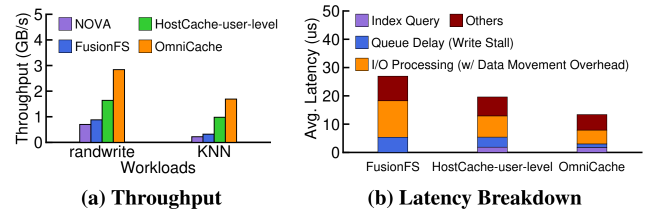
2.2 Limitations of State-of-the-art Systems

We next present the limitations of the state-of-the-art systems. Table 1 categorizes and compares current approaches with the proposed OmniCache.

Failure to Exploit Near-storage Memory for Caching: Existing near-storage designs [9, 31, 33, 44] either fail to exploit device memory or the combined capabilities of host and device memory [42]. CrossFS [31], FusionFS [9], and Insider [33] lack support for a host or device caching, while λ-IO [44] only utilizes host caches through the OS file system. This results in high I/O overheads, data movement, and frequent kernel traps. Leveraging DevCache and HostCache together presents new opportunities to accelerate performance.

RocksDB	MySQL	DiskANN
99.99%	93.32%	95.89%

Table 2: Unaligned I/O Requests Ratio.



(a) Throughput **(b) Latency Breakdown**

Figure 1: Motivation Analysis: (a) shows aggregated throughput for an I/O intensive random write and data processing application (KNN) with 32 threads. For random write, each thread accesses a private 4GB file (128GB total); For KNN, threads share a large 128GB file with each thread accessing a non-overlapping range; (b) shows the latency breakdown for random write.

Lack of Concurrent I/O and Data Processing Support: Existing near-storage designs lack support for concurrent I/O and data processing across host and device layers [9, 31, 33]. In these designs, I/O and processing operations are mostly offloaded to the device using fewer and less powerful device-level processors and limited memory [9, 31, 33]. Host-only (OS) caching solutions like λ-IO impose concurrency limitations. The use of OS cache incurs scalability bottlenecks from coarse-grained inode-level locking and suffers from eviction stalls when the host cache is full [44]. These limitations affect the performance of I/O and data processing [9, 31, 33, 44].

Lack of Dynamic Offloading Support: Several state-of-the-art near-storage designs lack the capability to dynamically decide whether to process on the host or the device and always offload (with only partial support for λ-IO). This leads to higher data movement, queuing delays, and compute bottlenecks. In addition, all existing designs lack a holistic approach to concurrently process data across host and device layers. We show that dynamic offloading and concurrent processing can significantly enhance performance.

2.3 Analysis

First, to motivate **the importance of leveraging host and device memory for caching** and to demonstrate the significance of OmniCache toward concurrent I/O and data processing operations. We compare OmniCache against state-of-the-art NOVA (a kernel file system) [43], FusionFS (a near-storage file system without caching support), and an extended host-only caching design using OmniCache (Figure 1a). Like prior systems [9], we use a machine with 512 GB DC Optane NVM for storage, 64 CPUs, and 32 GB DRAM. We set the total cache size to 20GB for all workloads. For OmniCache, we use a 16GB host cache and a 4GB device cache. For brevity, we focus on two workloads: (1) an I/O-intensive random-write

benchmark that generates a 128 GB file with random 1KB writes using 32 threads, and (2) an I/O + processing-intensive K-nearest neighbor search (KNN) that does not fit in memory, as used in prior research [44].

For the random-write workload, NOVA exhibits inferior performance due to system call and kernel software overheads and the lack of caching support. FusionFS also performs poorly due to the absence of caching. *HostCache-user-level* suffers from data movement and frequent write stalls during cache evictions. However, OmniCache significantly improves performance by leveraging collaborative host and device cache usage (§4.2). For data-processing intensive K-nearest neighbor (KNN), *HostCache-user-level* incurs high data movement costs as it exclusively processes data on the host. In contrast, OmniCache achieves higher performance gains by concurrently utilizing HostCache and DevCache for collaborative processing (§4.3).

Second, to highlight the necessity of exploiting near-storage RAM to optimize unaligned I/O requests, Table 2 shows the substantial unaligned I/O request ratios in popular real-world applications, including RocksDB, MySQL, and DiskANN[18]. All applications exhibit exceptionally high unaligned ratios, which motivates OmniCache's collaborative I/O caching, which aims to minimize data movement overhead associated with unaligned I/O requests (§4.2). In RocksDB, we observe that for 10 million keys and a 4KB value size (fillrandom and readrandom), over 99.99% of the 141 million total I/O requests were unaligned. This emphasizes the prevalence of high unaligned I/O requests in log-structured systems. Similarly, MySQL and DiskANN (a state-of-the-art approximate nearest neighbor search algorithm) also contend with a significant number of unaligned I/O requests.

Next, to **understand the software overheads**, we show the cost breakdown of these approaches in Figure 1b. We present the average latency breakdown of OmniCache for a random write (*randwrite*) workload. Firstly, the overhead of OmniIndex is notably low, accounting for only 12% of the total time in OmniCache. Secondly, OmniCache's capability in minimizing write stalls leads to a marked decrease in the queue delay overhead. Compared to *HostCache-user-level*, the queue delay overhead is significantly reduced from 18.10% to 9.3% with OmniCache due to its ability to reduce write stalls. Furthermore, OmniCache effectively reduces data movement overhead in I/O and data processing for unaligned I/O requests owing to its efficient near-cache I/O principle.

3 Goals and Overview

Motivated by the need to exploit host and device caches collaboratively for accelerating I/O and data processing, we next discuss design goals and overview of OmniCache.

3.1 Design Goals

OmniCache introduces a novel caching mechanism to harness the potential of both host and device-level memory for caching by leveraging the hardware and software capabilities of near-storage accelerators, host CPUs, and file systems distributed across the host and device layers. By combining their strengths and compensating for their weaknesses, OmniCache aims to achieve the following design goals:

Faster I/O using Near-cache I/O Principle: OmniCache introduces a new **near-cache I/O** principle that maximizes I/O operations to and from the nearest cache in both the host and device processors, thereby minimizing data movement. Near-cache I/O reduces data movement between the host and the device by only moving bytes actually requested by an application without requiring block-aligned data movement.

Collaborative Caching for Concurrent I/O: To address the lack of combined HostCache and DevCache use, one approach is to tier data between the caches. However, tiering hinders application threads from concurrently accessing the caches, leading to side effects like frequent CPU stalls during cache eviction. Besides, the smaller DevCache compared to HostCache complicates tiering. OmniCache addresses these challenges by supporting a horizontal paradigm that allows concurrent access to the caches. For concurrent access to non-conflicting blocks, a host-managed scalable index (OmniIndex) maps a range of blocks in a file to different caches.

Collaborative Data Processing with Distributed Caching: OmniCache exploits memory caching not just for I/O but to also accelerate data processing operations by reducing data movement between the host and the device. OmniCache achieves these goals by (1) creating mechanisms for collaborative data processing across host and device caches, (2) developing a model-driven approach to dynamically determine the optimal processing location (host or device) by leveraging hardware and software metrics (**OmniDynamic**), and (3) supporting concurrent data processing and merging results across host and device layers. OmniCache 's collaborative processing can benefit a variety of applications, especially those that involve processing and analyzing large-scale data in a parallel and distributed fashion. For example, graph processing, search engine for indexing and searching large webpage files, and NLP for extracting information and patterns from unstructured text.

Effectiveness on Byte-addressable CXLs: To minimize host-device data movement and associated queuing delays, OmniCache utilizes modern CXL technology to extend memory capacity and enable direct access to the accelerator's memory [1, 20]. Leveraging the host-managed OmniIndex, OmniCache uses CXL for direct DevCache access, thereby reducing additional data copies and queuing delays.

3.2 OmniCache Overview

We provide an overview of OmniCache components and briefly illustrate their functionalities. As shown in Figure 2, OmniCache comprises three main components: (1) a user-level library (OmniLib), (2) a user-level cache indexing structure (OmniIndex), and (3) a device manager (OmniDev). For this overview, we consider simple I/O (e.g., read(), write()) and data processing operations, e.g., read-CRC-write (read a

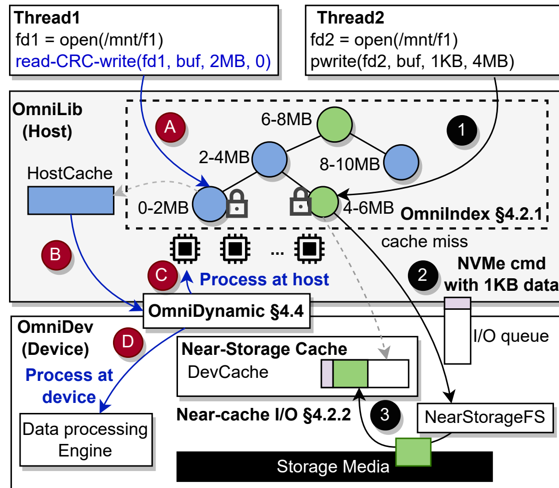


Figure 2: OmniCache High-level Design. Figure shows OmniCache concurrently handling I/O and data processing flows. For **I/O (black arrow),** ❶ application issues 1KB overwrite, which OmniLib intercepts, uses OmniIndex to locate the data in HostCache or DevCache. ❷ On a cache miss, the request is dispatched as an NVMe command using an I/O queue. ❸ OmniDev fetches the request, reads a 4KB block from storage to DevCache and updates the block. For **data processing operation (blue arrow),** Ⓐ application invokes OmniLib' read-CRC-write, which searches OmniIndex Ⓑ, uses a dynamic model to process in the host (Ⓒ) or the device (Ⓓ) or collaboratively on both.

4KB data block, calculate the checksum, and write it back).

OmniLib: The user-level component exploits host-level multicore CPUs, enables collaborative caching, and performs data processing. OmniLib performs various tasks such as dispatching I/O requests, managing cache resources, facilitating concurrent I/O and data processing operations, and handling data evictions.

Figure 2 illustrates the flow of operations across OmniCache 's components. When an application opens a file and initiates an I/O operation, OmniLib intercepts the system calls and creates a per-file I/O queue. Next, OmniLib converts all POSIX I/O calls to NVMe-like commands and adds them to the I/O queues for device (OmniDev) processing (shown from ❶ to ❸). OmniLib also handles data processing operations and offers predefined application interfaces such as *read-checksum-write* (Ⓐ) or *read-compress-write*. These operations are converted into a vector of NVMe commands and added to the I/O queue for either offloading to the device or processing at the host.

For caching, OmniCache divides the responsibilities across the host and the device layers. OmniLib provides the indexing for the HostCache and the DevCache, checks the presence of data using the index, and manages HostCache. OmniLib also decides when to evict from HostCache and DevCache by implementing a two-step LRU eviction.

For data processing, OmniLib implements a model-driven offloading engine to dynamically (Ⓑ) decide whether to offload processing (e.g., read-CRC-write) to the host (Ⓒ) or the device (Ⓓ). Finally, OmniLib also provides extensibility to use *CXL.mem* by directly copying data and commands to DevCache and avoiding queuing delays and data copies.

Fine-grained Indexing (OmniIndex): We implemented it as a part of OmniLib, providing scalable indexing for efficient data retrieval. OmniIndex locates the data stored in HostCache, DevCache, or storage. In addition to collaborative and concurrent use of the caches, it performs ownership management of block ranges in a file and data eviction. Figure 2 shows OmniIndex represented by a per-file range tree indexing structure. Each node corresponds to a specific range/segment of a file, with a pointer to the memory buffer in the HostCache or the DevCache, or the storage. Blue and green nodes in the figure indicate data residing in HostCache and DevCache, respectively. OmniIndex's fine-grained range locks handle concurrent I/O and processing requests across threads, ensuring conflict-free access across the host and the device. OmniIndex also tracks dirty data for cache eviction (§4.2.2).

OmniDev: The near-storage component consists of a file system, a data processing engine, and support for near-storage caching. OmniDev's file system is similar to the prior near-storage file system, comprising in-memory and on-disk metadata structures and journaling for crash consistency [9]. The data processing engine handles processing requests, retrieving them from the I/O queues, updating NVMe commands with the processed output, and setting a completion flag.

With respect to caching, OmniDev handles the allocation (space management) of the DevCache using a simple device-level memory manager. On a cache miss for an I/O or processing request, OmniDev allocates space within the DevCache using its internal memory allocator, processes the request, and returns the allocated cache block's address (a block number, see §5) to update the OmniIndex using OmniLib. The coordinated cache management between the OmniLib (host-level) and OmniDev (devel-level) provides efficient cache management and fast data lookup.

4 Design

We describe OmniCache's architecture, followed by its scalable approach to use and manage HostCache and DevCache for I/O and data processing operations.

4.1 Cache Architecture

OmniCache aims to minimize data movement and software overhead by performing I/O and data processing closer to the data, resulting in higher IOPS. It utilizes both host and device caches. We discuss the rationale for distributed cache management's placement, management, and challenges.

Host Caching in the Userspace: We implement HostCache allocation and management in the user-level OmniLib to avoid kernel traps and system call bottlenecks of an OS-level cache and customize cache admission and evictions for I/O and data processing. Each application is reserved with a configurable cache memory, which is managed by OmniLib. To support file sharing and the host (and device cache) across applications, OmniCache implements cache as a shared memory (§4.2.2).

Device-level Cache: Two important design considerations for DevCache are: (1) maintaining data exclusively vs. inclusively in the host and the device cache without increasing data movement and communication overheads; (2) ensuring that applications only with correct permissions access the cache blocks despite the direct I/O bypassing the OS. Regarding (1), *we employ an exclusive caching approach,* where data blocks are either stored in the HostCache or the DevCache or the storage (which results in a cache miss). We use exclusive instead of inclusive cache (where data could be duplicated in the host and device caches) for the following reasons: Firstly, exclusive cache avoids duplicate blocks across caches, unlike inclusive cache, increasing cache coverage. Secondly, an inclusive cache to maintain consistency can incur high communication costs between the host and the device. Finally, because of the significantly different HostCache (larger) and the DevCache (smaller) capacities, an exclusive cache provides the flexibility to vary the eviction frequencies.

Regarding (2), the OS and the OmniDev manages the permission checks and access control, which we inherit from prior systems [9, 31, 33]. Briefly, for a process to access a file, a per-file I/O queue is only created by the OS if the process has access permission to the file and the queue is also tagged with the credential by the OS. OmniDev, before dequeuing and dispatching a request, checks if the request in the I/O queue has the necessary permission to access/update the file's content before checking the DevCache and the disk.

4.2 Collaborative Caching for I/O

We first discuss the techniques employed by OmniIndex for fast indexing and locating cache blocks. We then elaborate on how OmniCache reduces data movement during various I/O operations by adhering to the near-cache I/O principle. Finally, we describe how collaborative caching enhances concurrent I/O by mitigating eviction stalls.

4.2.1 Scalable OmniIndex

We address the above challenges by designing OmniIndex, a scalable and highly concurrent cache indexing mechanism based on a range tree. OmniIndex indexes data in both HostCache and DevCache, and is managed only by the host (OmniLib). Managing OmniIndex exclusively in the host avoids communication and consistency overheads of maintaining OmniIndex between the host and the device and leverages multicore CPU parallelism for concurrent index lookup. It also provides the flexibility to customize OmniIndex and use it for cache admission and eviction based on the application and user requirements.

OmniIndex, a per file range tree, offers a unified view of the host and device caches and the storage. Each OmniIndex node represents a specific data range within the file, with blocks potentially residing in the HostCache, DevCache, or the storage. In Figure 2, blue-colored nodes indicate data in the HostCache, green nodes represent blocks in the DevCache, and all others represent blocks on the storage device. OmniCache utilizes OmniIndex to determine the data location.

The I/O or data processing requests are assigned to the device by the host, which uses the OmniIndex to locate existing data blocks, if not present in any caches, allocate and updates the OmniIndex with a new node. The device CPUs do not access or update the OmniIndex.

Concurrent Non-Conflicting Access: For concurrent access/updates to a file's non-conflicting data blocks by the host CPU threads, each node range in the OmniIndex is equipped with a read-write lock. Threads acquire per-range lock before accessing the corresponding data from the cache, performing I/O, or processing. When the data is not in the host and the device caches, a range lock is acquired before issuing an I/O command. We shortly discuss the details of using the OmniIndex to perform I/O (e.g., write, read) and processing.

Avoiding Conflicts: To prevent conflicting and concurrent updates of range by the host and the device CPUs, OmniCache employs a range-level ownership model, by assigning an ownership of a range to the host or the device. This is feasible because the host OmniLib is responsible for offloading I/O or processing requests to the device and ensures that only one entity (host or device) can modify the data within the node range at any given time and preserve data integrity.

Tracking Dirty Data for Persistence: OmniIndex is essential for managing data in the host and device. Each OmniIndex node includes a dirty bit for each range and a bitmap array to track block dirtiness within a range. To update the HostCache or DevCache, pages are allocated, and the OmniIndex is updated at the range level by setting a block's dirty bit in the range. Dirty bits are set for updates and cleared during file commits or flushes (e.g., `fsync`).

Memory Overheads of OmniIndex: The memory overhead of OmniIndex is minimal. For a 1TB file, regardless of the data location (host or device cache), the index only needs 128MB ($< 0.001\%$) of memory, with each OmniIndex node occupying 256 bytes. To reduce memory requirements further, one could have larger OmniIndex ranges or employ huge pages, which we will focus in the future.

4.2.2 I/O Operations with OmniCache

We next discuss a basic set of I/O operations, such as write and read, when using OmniCache, which mainly aims to perform near-cache I/O to avoid data movement.

Write: When an application performs write operations to expand a file, OmniCache employs the near-cache I/O principle. Initially, the data is written to the HostCache of a OmniIndex node, followed by updating the node's information, including updating OmniIndex node's dirty bit information. Furthermore, to reduce the depth of the OmniIndex and enable batch eviction, writes are merged into a single node with a maximum range of a pre-configured size (default is 2 MB). However, cache pages are allocated at the granularity of the block size. We will shortly discuss the concurrent and collaborative approach to update HostCache and DevCache.

Overwrite: To optimize overwrites, OmniCache follows a near-cache I/O approach to minimize data movement over-

head. If the blocks to be overwritten are already present in the HostCache or DevCache, OmniCache updates the cache and marks the corresponding range as dirty. In case of a cache miss, unlike existing designs, OmniCache avoids fetching the entire range of blocks from storage to the host. Instead, it only reads the relevant block(s) to the DevCache to reduce write amplification between the device and the host, applying changes directly to the blocks in DevCache and updating the OmniIndex. For instance, consider a scenario where an application issues a 1KB `write()` on a 4KB block not present in the cache. Other recent system designs must perform block-aligned reads from storage to the host, reading the entire 4KB block, resulting in I/O amplification and data movement cost. However, OmniCache leverages the benefits of the DevCache to overcome this limitation. The advantages of using DevCache are further demonstrated in §6.

Read: OmniCache first searches OmniIndex to locate the blocks, then reads the blocks from HostCache, DevCache, storage, or all. Read operations work like overwrites by loading the missing data block to DevCache if space is available, and only returning requested data to the application. Similarly to Linux, OmniCache identifies access patterns to enhance prefetch granularity (up to 2MB). Importantly, for blocks in multiple OmniIndex nodes or storage, the OmniIndex with fine-grained lock is used to concurrently read blocks, resulting in lower latency and higher throughput.

File Commit (fsync): OmniCache uses OmniIndex's range and per-block dirty bit to commit one or more blocks to storage. For blocks in DevCache, OmniCache creates and issues an I/O command, and OmniDev handles the file commit. The `fsync` is treated as a barrier operation on a file.

File Sharing: For sharing files across processes, OmniCache allocates cache pages within a shared memory region facilitated by our shared memory allocator. Access to the shared cache and OmniIndex is limited to processes with the necessary file permissions. In the case of processes with write permissions, OmniCache maps the shared memory as writable. However, like many prior user-level direct-access file system designs, the direct-I/O approach is susceptible to corruption [9, 21, 24, 25, 29, 33]. For instance, a process with write permission could potentially corrupt the OmniIndex. While prior near-storage designs [31] transition to the OS to handle shared file updates, we have also adopted an approach used by previous user-level file systems (such as Aerie [40], Strata [25], uFS [27]), which involves a trusted third-party server mediating access to the shared OmniIndex using lease-based locks. Nevertheless, our future work will optimize inter-process sharing, as this work primarily focuses on accelerating single multi-threaded applications.

4.2.3 Concurrent Caching and Reducing Eviction Stalls

Concurrent use of HostCache and DevCache and low-overhead cache eviction is crucial for minimizing application stalls and optimizing performance. However, the limited capacities of caches can result in frequent evictions for data-intensive applications, affecting performance [22].

To tackle these challenges, we propose collaborative caching and concurrent eviction. In collaborative caching, application threads concurrently use HostCache and DevCache, switching between them when one cache is full. This reduces compute stalls and enhances performance (see Section 6).

Two-step LRU Eviction: The effectiveness of dual-cache utilization depends on accelerated cache eviction. New data updates initially enter HostCache. When HostCache reaches capacity (no available space), OmniLib directs writes to DevCache and starts concurrent eviction in HostCache. A background eviction thread manages two levels of LRU information: (1) a file-level LRU, where all closed or inactive files are added to a global LRU list; (2) a per-file OmniIndex LRU list that tracks least-recently-used ranges. A file or a range becomes LRU if not accessed within a configurable 30-second epoch, akin to Linux. In the first step of the eviction process (from HostCache or DevCache), we evict LRU files. If free cache space drops to a lower threshold (10% free memory), applications continue inserting into the cache. Otherwise, the second step entails range-level LRU eviction, removing blocks from the device and host. For DevCache eviction, OmniLib sends an NVMe-like eviction command to OmniDev to evict ranges. The collaborative caching and two-step eviction ensure a seamless transition between HostCache and DevCache, concurrent utilization of different caches, reduced compute stalls, and improved performance (see Section 6).

4.3 Collaborative Processing with Caching

OmniCache also leverages HostCache and DevCache to accelerate data processing. The effectiveness of offloading operations for near-storage processing depends on the frequency of I/O operations and factors such as the impact of using less powerful on-device computing and smaller memory resources. We first discuss the application interface support, the challenges, and solutions to support collaborative processing.

Application Interface for Processing: Like prior approaches, OmniCache requires applications to use pre-defined processing functions [9, 33, 44] provided by OmniLib. Unlike λ-IO, which utilizes eBPF, imposing restrictions on operations/functions involving floating-point calculations and unbounded loops. OmniCache follows a near-storage file system paradigm; it borrows and extends the CISC-like interface from prior work [9], enabling application developers to offload richer operations/functions, such as checksum or compression. As shown in Figure 2, the I/O and data processing operations are converted and stacked as a vector of NVMe commands and offloaded as batched operations (referred to as CISC operation [9]). Unlike prior systems, OmniCache can collaboratively process simple and complex operations (e.g., *read-cal_distance-nearestK*) in the host, the device, or both.

Challenges: Collaborative data processing requires concurrent host-device processing, dynamic data migration across caches, and intelligent decisions on offloading based on hardware and software costs (e.g., data movement overheads, pro-

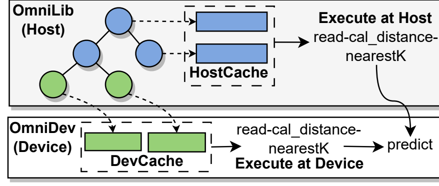


Figure 3: OmniCache Collaborative Processing for KNN. *read-cal_distance-nearestK* is concurrently executed across host and device.

cessing times, and queuing latency).

Key Ideas: To address these challenges, we extend collaborative caching approach, incorporating I/O caching and data processing. Firstly, we use OmniIndex for concurrent range-level data processing on both host and device with fine-grained range concurrency, enhancing performance. Secondly, we introduce a dynamic model that considers hardware and software metrics (e.g., storage, memory, compute time, queuing latency). This model continuously monitors the system and dynamically selects the best offloading location to use resources efficiently.

4.3.1 Extending OmniIndex for Compute Cache:

We improve OmniIndex by adding *processing buffer*, an intermediate computation buffer separate from cache buffer. Processing buffer is linked to each tree node and can be stored in host or device memory. Processing buffer is accessed via an address reference in each interval tree node. This helps OmniCache quickly find processing states, split and merge processing on the host and the device.

Case study: K-Nearest Neighbor Search (KNN): We demonstrate `KNN`, a widely used ML algorithm that identifies the K-nearest data points to a given query point. Since the dataset exceeds the memory capacity, `KNN` reads a large data chunk, calculates distances between data points, selects the K-nearest points, predicts classification based on them, and optionally writes results to a new file.

In OmniCache (see Figure 3), we handle this with a combined I/O and data processing operation called `read-cal_distance-nearestK`. This operation reads a data range, computes distances, and selects the K-nearest points. Intermediate results, like calculated distances, are stored in a processing buffer. OmniCache executes `read-cal_distance-nearestK` concurrently on both host and device, leveraging collaborative processing.

Afterward, OmniCache merges the K-nearest points for final classification prediction. It involves transferring data between host and device, requiring data copying. Importantly, `read-cal_distance-nearestK` and prediction operations can be performed on either host or device, determined by our resource-driven dynamic offloading strategy (§4.4).

4.4 Resource-driven Dynamic Offloading

OmniCache aims to increase near-cache and near-data processing on both host and device while minimizing data movement. However, disparate compute capabilities, cache capacity, and data transfer times between storage, HostCache, and DevCache necessitate a dynamic approach for determining the optimal processing location. The **challenges** in making these offloading decisions are threefold. First, processing operations where the data is distributed across host and device caches may incur data movement. Second, hardware and software metrics to decide where to offload, such as computational costs, memory requirements, and I/O frequency, can vary significantly based on the processing complexity. Third, monitoring both host and device hardware and software metrics without interfering with data-plane operations is critical.

Model-driven Approach: To address these challenges, we introduce **OmniDynamic**. It leverages a model-driven approach coupled with ongoing monitoring of both host and device resources. We begin by outlining the model and then detail our implementation approach. The model (Equation 1) estimates the processing time for each request and determines where the request will be processed (host or device).

T_h and T_d calculate the processing time for a request on the host and the device, respectively, by considering various factors: **Data Ratio** (R) represents data associated with a request distributed across HostCache, DevCache, and storage. The ratios R_{hm}, R_{dm}, and R_s represent the portion of data in the host memory (hm), device memory (dm), and storage (s) for each request. **Execution Time** (E) captures the processing cost alone, Eh_{avg} represents the average time to execute a request on the host, while Ed_{avg} represents the average time on the device. **Data Transfer Cost** (B) captures the data movement between HostCache, DevCache, and storage. B_{hm_dm} denotes the data transfer bandwidth between HostCache and DevCache, B_{ds_hm} represents the bandwidth between storage and HostCache, and B_{ds_dm} represents the bandwidth between storage and DevCache. Finally, **Queue Latency** represents the completion time of a request, which depends on the queuing delay. This varies based on the number of I/O and data-processing requests in the per-file I/O queue and the average time required to process a request ($Cmd_{avg} * Q_{len}$). The queue delay increases during cache eviction.

$$\begin{aligned} T_h &= R_d D/B_{hm_dm} + R_s D/B_{ds_hm} + Eh_{avg} \\ T_d &= R_h D/B_{hm_dm} + R_s D/B_{ds_dm} + Cmd_{avg} * Q_{len} + Ed_{avg} \end{aligned} \tag{1}$$

Continuous and Low-interference Monitoring: To realize the dynamic offloading model, OmniCache continuously monitors hardware and software parameters across the host and device layers using the OmniDynamic component. This component operates at the intersection of OmniLib and OmniDev, collecting metrics such as cache data ratios, data movement bandwidths, processing costs, and queue wait-time overheads.

For metric collection at the device, we extend each NVMe command for data processing with additional counters, including on-device processing time (Ed), data movement bandwidth between DevCache and HostCache (B_{hm_dm}), and between storage and DevCache (B_{ds_dm}).

As depicted in Algorithm 1, the initial phase is devoid of resource parameters and relies on OmniIndex to manage data distribution across HostCache, DevCache, and storage.

Algorithm 1: Model-driven Data Processing

```
1  All measurements are done periodically (per epoch)
2  Query OmniIndex to get data distribution ratio (R_h and R_d)
3  Get current queue length from I/O queue (Q_len)
4  Compute T_h and T_d based on Equation 1
5  if T_h <= T_d then
6  |   load_data_to_host()
7  |   measure_devmem_to_host_bw(&B_hm_dm)
8  |   measure_devstorage_to_host_bw(&B_ds_hm)
9  |   execute_request_at_host()
10 |   measure_avg_execution_latency(&Eh_avg)
11 else
12 |   move_data_to_device()
13 |   measure_host_to_devmem_bw(&B_hm_dm)
14 |   measure_devstorage_to_devmem_bw(&B_ds_dm)
15 |   execute_request_at_device()
16 |   measure_avg_execution_latency(&Ed_avg)
17 |   measure_queue_latency(&Cmd_avg)
```

By leveraging the cache ratios (R), OmniDynamic makes offloading decisions with a preference for near-data processing. When data exclusively resides on the host or the device, it undergoes processing without requiring data movement. Conversely, data scattered across caches and storage prompts data transfer, typically from the layer with a smaller data footprint.

OmniDynamic calculates the average processing times (Eh_{avg} and Ed_{avg}) for each request type by tracking the processing cost in the host or device. To measure the time spent by requests on the per-inode I/O queue ($Cmd_{avg} * Q_{len}$), OmniLib updates the queue admission time, while OmniDev updates the request completion time in the per-inode I/O queue.

4.5 Exploring CXL Extensibility with OmniCXL

To explore OmniCache's potential with emerging technologies like CXL.mem for caching, we introduce OmniCXL. As explained in §2.1, CXL.mem enables direct host access to an accelerator's memory. We do not assume hardware-supported cache coherence between HostCache and DevCache. In this context, we investigate how OmniCache leverages CXL.mem to reduce data copy and queuing bottlenecks while ensuring safe operation without hardware-level cache coherence support. In OmniCache's queue-based design, all requests incur overheads like packing and copying data and NVMe commands from the application (or device) buffer to the DMA-enabled I/O queues, queuing delays, and host CPU overheads like polling for request completion.

Reducing I/O and Processing Overheads: To reduce the above overheads, we propose extending OmniCache to leverage *CXL.mem* (OmniCXL). In this approach, device memory appears as an additional NUMA node in the host OS, a widely used abstraction for memory expansion. To use the device memory as DevCache, the OmniLib of a process memory maps and registers a designated region within the address space as DevCache. The DevCache size for each application is determined based on a specified limit.

When an application issues I/O operations like `write()` that cannot be cached in the HostCache due to space constraints, OmniLib directly writes data to DevCache after acquiring the range lock in OmniIndex and flushes the data to memory. This enables OmniCXL to avoid (1) copying data between application and device queues, (2) packing (at host) and unpacking (at device) NVMe commands, (3) reducing queuing delays, and (4) continuous polling for request completion, thereby minimizing CPU overheads. Furthermore, data copy overheads are avoided for processing requests offloaded to the device (e.g., `append-checksum-write`), but the request queuing and polling are still necessary.

5 Implementation Details

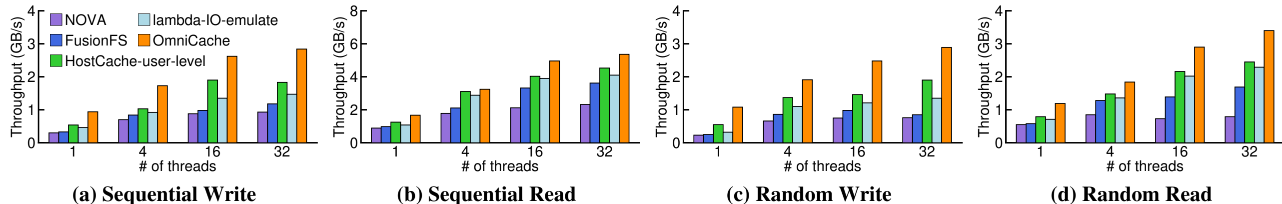
We first describe the implementation details at the near-device and host layers, followed by our approach to emulate near-storage and CXL.mem.

First, we implement an emulated **OmniDev** as a device driver (8K LOC) due to the lack of access to a programmable storage hardware, similar to prior work [9, 31]. To understand OmniCache's impact on faster and slower storage, we implement two distinct near-storage backends: one on Intel Optane Persistent Memory (PM) by extending PM file system and the other on NVMe-based SSDs that uses block-level ext4 with I/O operations bypassing the OS cache. We also add a storage processing engine to OmniDev.

Second, to manage DevCache, in OmniDev, we implement a lightweight and efficient memory allocator that uses a bitmap array to track the availability of cache blocks. The allocator returns a block ID to host-level instead of exposing the device's memory address to the host.

Third, OmniLib (discussed in in §3.2) uses a shim library to intercept POSIX I/O operations and convert them to OmniDev compliant NVMe commands. For HostCache management, we extend the scalable *jemalloc*[4] allocator for cache block allocation and release. When using a non-CXL near-storage device, the NVMe commands are copied to the per-file NVMe queues, which are later de-queued and dispatched by the OmniDev. In contrast, for *CXL.mem*, OmniLib directly accesses the device memory. We implement a CXL memory allocation semantics for OmniLib to register and allocate a CXL namespace that is shared across OmniLib and OmniDev.

Finally, **to emulate device memory and compute speeds, and CXL.mem latency and bandwidth,** we employ a two-step emulation. First, to emulate a slower device memory access from the host CPUs, we map device memory on a NUMA socket (node) remote to the host CPUs but local to the device CPUs. Precisely, in a system with two sockets, we allocate the host memory on NUMA *node*0, which is local to the host CPU and remote to the device CPUs. On the contrary, we allocate device memory on NUMA *node*1, which is local to the device CPU but remote to the host CPU. Next, to emulate device memory bandwidth, we throttle device memory's bandwidth using thermal throttling [6], lower device CPU speeds using frequency scaling, and add software latency to vary PCIe latency. In §6.4, we study the sensitivity of OmniCache on these hardware parameters.



(a) Sequential Write (b) Sequential Read (c) Random Write (d) Random Read

Figure 4: Microbenchmark. Threads use private files; workload size fixed at 64GB with 1KB I/O size and total cache size is 20GB.

6 Evaluation

We evaluate OmniCache to answer the following questions:

- How effective is OmniCache's collaborative use of HostCache and DevCache to improve I/O performance?
- Can OmniCache accelerate data processing with its concurrent and model-driven dynamic offloading?
- How does using OmniCache in conjunction with CXL impact the performance?
- What is the overall impact on real-world applications?

6.1 Experimental Setup

Environment: We use a dual-socket, 64-core, 2.7GHz Intel(R) Xeon(R) Gold platform with 32GB memory. For storage, we use a 512GB (4x128GB) Optane DC persistent memory with App-Direct (persistent) mode to represent the upcoming fast storage as well as 512GB NVMe SSD to study the benefit of OmniCache on slower storage. (see appendix appendix A.2.1). We emulate DevCache with 4GB of DRAM for caching and for OmniDev, we reserve 4 CPUs.

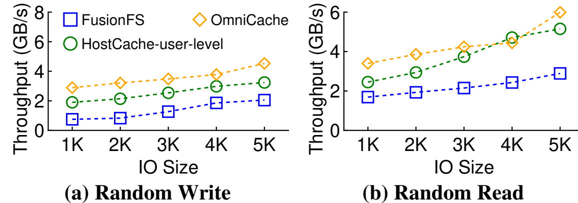
Methodology: For comparison, we consider the state-of-the-art PM OS file system, **NOVA**, and the near-storage file system, **FusionFS**, which lacks caching support (***FusionFS***). To understand the benefits and implications of host-only caching, we explore two configurations: (1) user-level host cache with OmniIndex atop FusionFS, referred to as ***HostCache-user-level***; (2) emulated λ-I/O without FPGA but with OS caching (***lambda-IO-emulate***), which does not exploit near-storage cache (see Table 1). We emulate λ-I/O due to the unavailability of a customized hardware platform (Daisy/DaisyPlus OpenSSD) and OS (PetaLinux), and significant engineering challenges as highlighted by the authors [5]. Moving on to the OmniCache configuration, we begin by comparing our PCIe-based implementation without OmniDynamic, where offloading is solely determined by the data presence ratio (***OmniCache***). Subsequently, we compare ***OmniCache-dynamic*** to emphasize the impact of OmniDynamic on data processing performance. Finally, we evaluate ***OmniCXL*** to demonstrate the effect of CXL on performance. For all evaluation results, the total cache size is kept the same.

6.2 I/O Performance

We first evaluate I/O performance using sequential and random access I/O patterns. We vary workload threads from 1 to 32, each issuing 1KB I/O requests, resulting in a total workload size of 64GB. Figure 4 shows the cumulative throughput for private file access without file sharing, and Table 3 shows the benefits of using OmniIndex fine-grained concurrency when sharing files with multiple threads. Regarding caching, the *HostCache-user-level* and *lambda-IO-emulate* approaches employ a total of 20GB of host DRAM cache, while *OmniCache* configurations use 16GB HostCache and 4GB DevCache, a configuration used in prior systems [9, 33].

	FusionFS	HostCache-user-level	OmniCache
Readers	978	1893	2323
Writers	523	1223	1732

Table 3: File sharing. Results show aggregated throughput (MB/s) for 16 reader and 16 writer threads on a shared 64GB file.



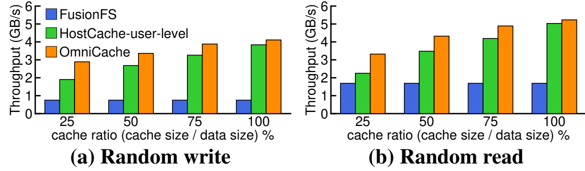
(a) Random Write (b) Random Read

Figure 5: I/O Size Study.

Observation: Figure 4a and 4c show results for sequential and random write workloads, respectively. As anticipated, NOVA and *FusionFS* lack caching and access storage for all I/O operations, which results in poor performance. Both *HostCache-user-level* with user-level caching and *lambda-IO-emulate* with OS caching show improvements but face high I/O stalls due to frequent cache eviction, particularly for random workloads. Similarly, for read workloads shown in Figure 4b and 4d, the host caching approaches, for each 1KB request, fetch 4KB blocks, increasing data movement cost, host-cache pollution, and suffer eviction stalls.

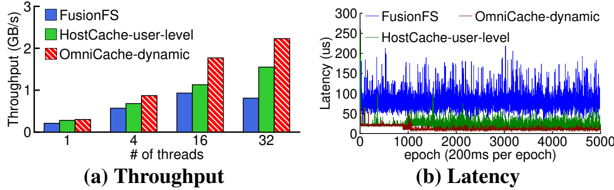
In contrast, OmniCache outperforms others by employing a collaborative approach that exploits host and device caches. Firstly, it adheres to the near-cache I/O principle, significantly reducing data movements between storage and host memory or between the host and device memory. For sequential access, DevCache identifies sequential access patterns, akin to Linux VFS, and preloads the entire 2MB data range (OmniIndex) to optimize data locality. However, it only returns the requested data (1KB) to the application, preventing I/O amplification and unnecessary data transfers. Subsequently, requests to the same 2MB range often result in cache hits. Secondly, OmniCache's collaborative caching performs writes or reads on both HostCache and DevCache, effectively minimizing write stalls and read times. As a result, OmniCache consistently outperforms *FusionFS* and *HostCache-user-level*, achieving performance gains of up to 2.53x and 1.52x, respectively.

I/O Size Sensitivity: We evaluate the impact of different I/O sizes by varying the request size from 1KB to 5KB while maintaining constant cache and workload sizes. This range encompasses both block-aligned and non-block-aligned



(a) Random write (b) Random read

Figure 6: Cache Sensitivity Study. 32 microbenchmark threads with different cache ratio from 25% to 100% (maintaining equal total cache size for *HostCache-user-level* and *OmniCache*).



(a) Throughput (b) Latency

Figure 7: I/O + Data Processing (Read-CRC-Write)

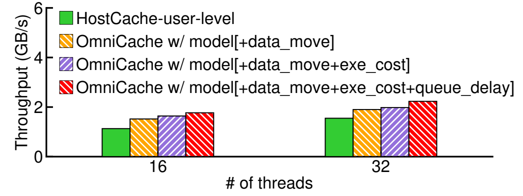
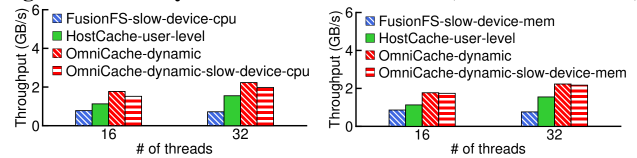


Figure 8: OmniDynamic Model Breakdown (Read-CRC-Write)



(a) Device CPU Frequency (b) Device Memory Bandwidth

Figure 9: Model's Sensitivity for Read-CRC-Write

requests, mirroring real-world application behavior (Table 2). For instance, in RocksDB, numerous application I/O requests are not block-aligned. Figure 5 illustrates that *OmniCache* consistently outperforms other approaches for non-block-aligned requests across various workloads, reaping the benefits of near-cache I/O. For block-aligned (4KB) random write, *OmniCache* provides performance gains, attributable to its concurrent I/O and collaborative caching that reduces write stalls. For block-aligned random reads (e.g., 4KB), *OmniCache* performs similar to *HostCache-user-level*, as it moves the entire block to the host.

Impact of Data to Cache Size Ratios: We investigate the impact of the data-to-cache size ratio on the throughput of random read and write workloads. Figure 6 displays this impact, with the x-axis ranging from 25% to 100% cache ratio. A 25% ratio implies that the data size is four times larger than the combined HostCache and DevCache sizes. At lower ratios, *HostCache-user-level* experiences frequent evictions and thread stalls. In contrast, OmniCache smoothly transitions application threads to utilize DevCache when HostCache reaches its capacity. It then initiates background eviction of HostCache before switching back to HostCache. Even at a 100% cache ratio, OmniCache outperforms others by minimizing data movement using near-cache I/O principle.

6.3 Data Processing with OmniCache

We evaluate the effectiveness of collaborative processing and dynamic offloading with OmniCache for I/O and compute-intensive read-CRC-write workload. Thread randomly reads 4KB data blocks, calculates checksum, and writes it back.

Throughput Analysis: Figure 7a illustrates that *FusionFS* encounters NVMe command communication and data copy overheads. It necessitates offloading each request to the device, performing computations on the device, and subsequently writing the data back to device storage. On the other hand, *HostCache-user-level* operates more efficiently when requests are served from the host cache, enabling direct execution on the host. However, in cases of cache misses, *HostCache-user-level* incurs data movement overhead between storage and host memory, which hinders computation.

In contrast, *OmniCache-dynamic* dynamically offloads data processing operations by efficiently considering multiple factors. Figure 7a illustrates that *OmniCache-dynamic* significantly enhances performance, especially under heavy workload scenarios. This improved performance results from its model-driven approach, which continuously monitors execution time, queue latency, and other factors to dynamically determine the optimal offloading location.

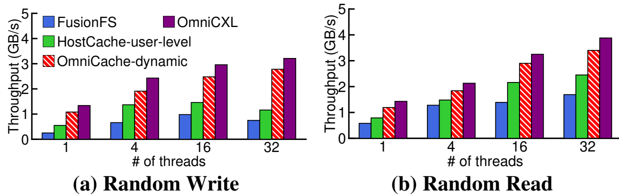
Latency Analysis: In Figure 7b, we examine latency variations in the read-CRC workload. HostCache handles write requests well initially but experiences fluctuations and delays due to evictions after epoch 1000, resulting in longer queue delays. Additionally, execution costs vary due to higher processor cache misses and I/O frequency. In contrast, *OmniCache-dynamic*'s model-driven approach, which considers factors like data distribution ratio, queue length, and execution costs, maintains lower and stable latency, outperforming *HostCache-user-level* by up to 1.42x.

6.4 OmniDynamic Model Effectiveness

We empirically validate the effectiveness of OmniDynamic model by deciphering the impact of model parameters and the sensitivity to hardware speeds.

Model Performance Breakdown: Figure 8 shows performance analysis by gradually incorporating different parameters of the model (data movement, execution time, queuing delays) and understanding their impact on the read-CRC-write workload. First, the decision to offload to device or process in host, is significantly influenced by data fraction (R_h or R_d) in HostCache or DevCache (*model[+data_move]*). Second, the execution time (Ed_{avg} or Eh_{avg}) fluctuates, impacted by factors like data presence in the host and device processor caches that can accelerate execution (*model[+data_move+exe_cost]*). Finally, overheads of queuing delay ($Cmd_{avg} * Q_{len}$) becomes particularly pronounced for higher application thread count and frequent background eviction (*model[+data_move+exe_cost+queue_delay]*).

Sensitivity to Hardware Parameters: To understand the *OmniCache-dynamic* model's sensitivity toward device CPU speed and low-bandwidth memory, we show the performance of OmniCache with slower CPU (Figure 9a). Due



(a) Random Write **(b) Random Read**

Figure 10: OmniCache with CXL

Vary Device Hardware Parameters			
Device CPU Frequency	2.7 GHz	2.0 GHz	1.2 GHz
Device Memory B/W	120 GB/s	60 GB/s	16 GB/s
PCIe Latency	900ns	1200ns	1500ns
# of ops. executed in host	**12.52M**	**14.67M**	**15.24M**
# of ops. executed in device	**4.19M**	**1.90M**	**834.32K**

Table 4: OmniDynamic Model's Sensitivity Analysis

to space constraints, we only consider *FusionFS* and OmniCache. The device CPU is throttled to 1.2GHz for FusionFS (*FusionFS-slow-device-cpu*) and OmniCache (*OmniCache-dynamic-slow-device-cpu*) compared to 2.7GHz for host CPUs. Similarly, in Figure 9b, the device memory is throttled to 16GB/s for FusionFS (*FusionFS-slow-device-mem*) and OmniCache (*OmniCache-dynamic-slow-device-mem*) compared to 120GB/s for host DRAM (8× bandwidth reduction). Despite slower device CPUs and reduced memory bandwidth, OmniCache provides up to 1.22x gains over HostCache by dynamically distributing work across the host and the device.

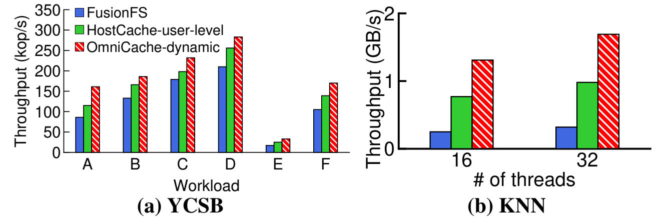
To comprehend the model's sensitivity and work distribution, in Table 4, we vary the device CPU, memory, and PCIe latency values and monitor *OmniCache-dynamic*'s offloading decision. As we gradually reduce the device CPU frequency, lower the memory bandwidth, and increase PCIe latency, *OmniCache-dynamic* increases operations on the host rather than indiscriminately offloading them to the device.

6.5 CXL.mem enabled OmniCache

Figure 10 shows the benefits of using CXL.mem for random access workloads. While OmniCache without CXL incurs overheads from data copies between the host and the device and queuing delays, *OmniCXL* directly accesses DevCache with CPU loads/stores after acquiring ownership of a range. This improves performance by diminishing these overheads. Furthermore, *OmniCXL* reduces CPU polling cost for request completion, all leading to 2.76x and 1.21x gains over *HostCache-user-level* and *OmniCache*, respectively.

6.6 Real-World Applications

We next evaluate the benefits of OmniCache on real applications. **LevelDB:** LevelDB is a widely-studied LSM-based persistent key-value store [3]. We modify LevelDB's *append→checksum→write* sequence by introducing the append-CRC-write operation and *read→checksum* sequence with read-CRC, similar to *FusionFS* (11 LOC changes). The checksum operations are used in LevelDB to avoid frequent commits on SST files [9]. We run experiments using the widely-used YCSB cloud workload [14] that comprises six distinct access patterns (A-F), each with differing read/write ratios and exhibits a Zipfian distribution with access locality.



(a) YCSB **(b) KNN**

Figure 11: Real-World Applications.

We use 512B value sizes, 40 million keys, and 32 threads.

Performance: As shown in Figure 11a, OmniCache outperforms across all workloads. Particularly, write-intensive A and F workloads show maximum gains over *HostCache-user-level* attributed to (1) near-cache I/O that reduces data movement for non-block aligned requests, (2) collaborative caching that minimizes CPU stall time, and (3) dynamic offloading to effectively use of host and device resources. Over FusionFS, OmniCache shows up to 1.92x gains. db_bench [8] for random workloads show even higher gains (see appendix A.2.2).

Nearest Neighbor Search (KNN): Next, we evaluate OmniCache using a complex KNN workload, utilizing an implementation from prior work [33]. However, we deviate from their assumption of the entire workload fitting into device memory. Instead, we employ a 128GB workload, with 20GB *HostCache-user-level* or 16GB HostCache and 4GB DevCache for OmniCache. To handle datasets larger than the cache size, the application divides the file into shards and, for each shard, performs distance calculations (see §4.3), merging the per-shard distances for KNN prediction.

As Figure 11b shows, *FusionFS*, without caching, performs poorly as KNN execution requires reading each shard from storage to the device. *HostCache-user-level* offers marginal improvement but encounters significant data movement and eviction overheads as the data size surpasses the cache. In contrast, OmniCache effectively and concurrently utilizes both host and device for *read-cal_distance-predict*, resulting in performance gains of up to 5.15x over FusionFS.

7 Conclusion

We develop OmniCache, a collaborative caching design to leverage host and device memory as cache to accelerate I/O and data processing. OmniCache achieves this through scalable indexing, concurrent caching and processing support, and a dynamic model-centric offloading technique leading to substantial performance gains on both microbenchmarks and applications.

Acknowledgements

We thank Youngjin Kwon (our shepherd) for the insightful comments to improve the quality of this paper. We also thank anonymous reviewers and the members of RSRL for their valuable feedback. This research was supported by funding from NSF CNS grants 1910593, 2231724, and Samsung Memory Solutions Lab. This work was carried out on the experimental platform funded by NSF II-EN grant 1730043.

References

[1] CXL Specification. https://www.computeexpresslink.org/download-the-specification.

[2] Gluster. https://www.gluster.org/.

[3] Google LevelDB . http://tinyurl.com/osqd7c8.

[4] jemalloc. https://jemalloc.net/.

[5] λ-IO GitHub. https://github.com/thustorage/lambda-io#building-and-running.

[6] memhog - Allocates memory with policy for testing. https://man7.org/linux/man-pages/man8/memhog.8.html.

[7] rbtree based interval tree as a prio_tree replacement. https://lwn.net/Articles/509994/.

[8] RocksDB db_bench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools/.

[9] FusionFS: Fusing I/O operations using CISCOps in firmware file systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 297–312, Santa Clara, CA, February 2022. USENIX Association.

[10] Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. Onyx: a protoype phase change memory storage array. In *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*, HotStorage'11, Portland, OR, 2011.

[11] ARM. https://www.arm.com/solutions/storage/computational-storage.

[12] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 121–134, Renton, WA, July 2019. USENIX Association.

[13] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 29–41, Santa Clara, CA, February 2020. USENIX Association.

[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*, pages 143–154, 2010.

[15] Jaeyoung Do, Victor C. Ferreira, Hossein Bobarshad, Mahdi Torabzadehkashi, Siavash Rezaei, Ali Heydarigorji, Diego Souza, Brunno F. Goldstein, Leandro Santiago, Min Soo Kim, Priscila M. V. Lima, Felipe M. G. França, and Vladimir Alves. Cost-effective, energy-efficient, and scalable storage computing for large-scale ai applications. *ACM Trans. Storage*, 16(4), October 2020.

[16] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM.

[17] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 173–187. USENIX Association, July 2020.

[18] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems*, 32, 2019.

[19] Myoungsoo Jung. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pages 45–51, 2022.

[20] Myoungsoo Jung. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, page 45–51, New York, NY, USA, 2022. Association for Computing Machinery.

[21] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a True Direct-access File System with DevFS. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 241–255, Berkeley, CA, USA, 2018. USENIX Association.

[22] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, Boston, MA, July 2018. USENIX Association.

[23] Hyojun Kim, Sangeetha Seshadri, Clement L Dickey, and Lawrence Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 33–45, 2014.

[24] Jinhyung Koo, Junsu Im, Jooyoung Song, Juhyung Park, Eunji Lee, Bryan S. Kim, and Sungjin Lee. Modernizing file system through in-storage indexing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 75–92. USENIX Association, July 2021.

[25] 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, 2017.

[26] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[27] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 819–835, New York, NY, USA, 2021. Association for Computing Machinery.

[28] NVIDIA Mellanox BlueField DPU. https://www.mellanox.com/files/doc-2020/pb-bluefield-smart-nic.pdf.

[29] 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 Conference on Operating Systems Design and Implementation*, OSDI'14, Broomfield, CO, 2014.

[30] Nicholas Rauth. A network filesystem client to connect to SSH servers. https://github.com/libfuse/sshfs.

[31] Yujie Ren, Changwoo Min, and Sudarsun Kannan. {CrossFS}: A cross-layered {Direct-Access} file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 137–154, 2020.

[32] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, page 62–73, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[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 (USENIX ATC 19)*, pages 379–394, Renton, WA, July 2019. USENIX Association.

[34] Samsung. Samsung Key Value SSD. https://www.samsung.com/semiconductor/global.semi.staticSamsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf.

[35] ScaleFlux. https://scaleflux.com/.

[36] Seagate RISC-V storage solution. https://www.seagate.com/innovation/risc-v/.

[37] Yasas Seneviratne, Korakit Seemakhupt, Sihang Liu, and Samira Khan. Nearpm: A near-data processing system for storage-class applications. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 751–767, New York, NY, USA, 2023. Association for Computing Machinery.

[38] SNIA. SNIA Computational Storage Technical Work Group (TWG).

[39] Mohammadreza Soltaniyeh, Veronica Lagrange Moutinho Dos Reis, Matt Bryson, Xuebin Yao, Richard P. Martin, and Santosh Nagarakatte. Near-storage processing for solid state drive based recommendation inference with smartssds®. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ICPE '22, page 177–186, New York, NY, USA, 2022. Association for Computing Machinery.

[40] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, Amsterdam, The Netherlands, 2014.

[41] Matthew Wilcox and Ross Zwisler. Linux DAX. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[42] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. Recssd: near data processing for solid state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 717–729, 2021.

[43] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, Santa Clara, CA, 2016.

[44] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. $\lambda - io$: A unified IO stack for computational storage. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 347–362, Santa Clara, CA, February 2023. USENIX Association.

[45] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. Spitfire: A three-tier buffer manager for volatile and non-volatile memory. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 2195–2207, New York, NY, USA, 2021. Association for Computing Machinery.

	FusionFS w/ NVMe Storage	HostCache-user-level	OmniCache
Random Read	731	2153	2521
Random Write	990	1573	1982

Table 5: Microbenchmark on NVMe Storage. Results show aggregated throughput (MB/s) for 32 benchmark threads.

A Appendices

A.1 Discussion

We have implemented various functions, including checksum, compression, nearest-neighbor search, and text search operations (not all shown due to space limitations). However, it is important to note that, like most existing systems, OmniCache assumes that OmniDev already incorporate these functions. Even previous near-storage systems [9, 10, 33], and those that use eBPF-based offloading [44] require device-level modifications and frequent kernel traps, which are not ideal for I/O-intensive applications. Developing a more generic offloading mechanism without requiring application changes requires compiler support, and is a complex task that falls outside the scope of this paper. In addition, our future work will explore using CXL.mem to enable memory-mapping (`mmap()`) support. To the best of our knowledge, existing systems also lack this particular feature.

A.2 Additional Performance Evaluation

A.2.1 Sensitivity to Slower NVMe Storage

To understand the impact of OmniCache when using slower storage media, NVMe-based SSD, we use the same experimental configuration and microbenchmarks to study the throughput. As shown in Table 5, OmniCache shows an even higher performance gain of 3.24x over *FusionFS* and 1.21x over *HostCache-user-level*. Notably, the gains are high compared to PM-based storage. These gains highlight the importance of collaborative cache use for slower storage. We observe performance gains even for data processing workloads (not shown due to space constraints).

A.2.2 Impact of OmniCache for LevelDB's db_bench

In order to assess the efficiency of the collaborative caching design offered by OmniCache, we also evaluate the random write and random read workload in Figure 12 using the widely-used db_bench for 1 million key-value pairs and 32 application threads with 4KB value size. OmniCache delivers higher performance across all workloads. These enhancements can be attributed to collaborative caching for I/O operations. In addition, OmniCache's dynamic offloading further amplifies these gains by ensuring optimal resource utilization between the host and device.

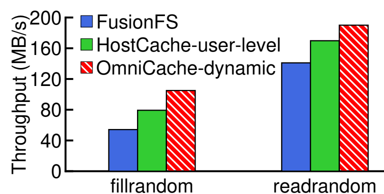


Figure 12: LevelDB (db_bench)

B Artifact Appendix

Abstract

The OmniCache artifact is the practical implementation of the collaborative caching design presented in this paper, aimed at optimizing I/O and data processing by utilizing both host and device memory as caches.

Scope

The artifact enables validation of the benefits of collaborative caching for concurrent I/O, the impact of dynamic model-driven offloading on data processing, and showcases the extensibility of OmniCache with CXL. OmniCache is licensed under Apache License 2.0.

Contents

The OmniCache artifact comprises user-level library (OmniLib) and the OS component for emulating near-storage device (OmniDev), which are required for the execution. The artifact comprises real-world applications (LevelDB, YCSB workload and KNN) besides microbenchmarks, as shown in the paper. The artifact includes steps to compile the user-level library, the OS, microbenchmarks, and applications, and steps to run these workloads.

Hosting

The artifact is available on GitHub: omnicache-fast24-artifacts.

Requirements

Our artifact is based on Linux kernel 4.15.18 and it should run on any Linux distribution. The current scripts are developed for Ubuntu 18.04.5 LTS. Porting to other Linux distributions would require some script modifications. Our artifact requires a machine equipped with Intel Optane memory.

Evaluation

We provide a comprehensive step-by-step README on GitHub to reproduce the experiment in the paper. As a brief overview of the evaluation, we illustrate how to execute a simple microbenchmark with OmniCache. More evaluations can be found on our GitHub page.

Before running, we assume the modified kernel (OmniDev) is installed, NearStorageFS is mounted on the machine (please see the README file) and the current work directory is in the project's root directory.

1. First, compile and install the user-level library (OmniLib):
```
$ source ./scripts/setvars.sh;
$ cd $LIBFS;
$ source scripts/setvars.sh
$ make && make install
```
2. Next, to run a simple micro benchmark:
```
$ cd $BASE/libfs/benchmark/;
$ mkdir build && make
$ ./scripts/run_omnicache_quick.sh
```

Expect output will be similar to "Benchmark takes 0.97 s, average throughput 4.45 GB/s". If the output matches the above, your environmental settings are appropriately configured.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

Symbiosis: The Art of Application and Kernel Cache Cooperation

Yifan Dai, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau
University of Wisconsin–Madison

Abstract. We introduce *Symbiosis*, a framework for key-value storage systems that dynamically configures application and kernel cache sizes to improve performance. We integrate Symbiosis into three production systems – LevelDB, WiredTiger, and RocksDB – and, through a series of experiments on various read-heavy workloads and environments, show that Symbiosis improves performance by 1.5× on average and over 5× at best compared to static configurations, across a wide range of synthetic and real-world workloads.

1 Introduction

Key-value storage engines, such as LevelDB [23], RocksDB [50], and WiredTiger [51], are essential components in modern data-intensive applications. These systems are deployed in numerous settings, including underneath relational databases [32, 52, 56], distributed storage systems [3, 18], graph processing engines [6, 13, 31, 55], stream processing systems [4, 12], and machine learning pipelines [30, 70].

A key factor in the performance of key-value storage systems is the effectiveness of in-memory caching. Unlike the traditional database approach [63], in which raw devices or other “direct I/O” mechanisms are employed to avoid file system caching, today’s key-value storage systems are often built on top of the file system, and thus (by default) will cache (compressed) data in the file system page cache. Furthermore, modern storage engines implement additional application-level caching structures (where data is cached in uncompressed form). The effectiveness of these combined caches can dramatically affect overall performance; proper usage can improve performance by an order of magnitude.

Unfortunately, this two-level structure greatly complicates performance tuning. How large should the application (uncompressed) cache be? How much memory should be dedicated to kernel-level (compressed) caching? The proper answer to this question requires sophisticated knowledge of workload, machine configuration, OS behavior, compression costs, and other relevant details; as workloads change over time, the answer too may change.

In this paper, we introduce *Symbiosis*, a system to coordinate application and kernel caches to maximize performance. The core component is an online approximate simulator used by a key-value store directly to adapt the size of the user-level cache. The simulator uses a modified form of ghost caching [19] to predict how different sized application caches will perform; Symbiosis uses these simulation results to periodically adjust the size of the application cache, thus improving performance. The online simulation includes novel optimizations to lower space overheads and handle nuanced kernel behaviors (such as prefetching), and guardrails to protect against unmodeled corner-case behaviors.

We show the utility of Symbiosis by incorporating it into three different key-value storage systems: LevelDB, WiredTiger, and RocksDB. Most of our work focuses on LevelDB, a popular LSM-based key-value storage system from Google [23]; through careful re-use of existing code (where appropriate), our modifications add roughly 1K lines to the code base. Across a range of read-heavy workloads, we show that Symbiosis improves LevelDB performance significantly (greater than 5×) as compared to unmodified LevelDB. We also show that our approach adapts effectively to workload changes and that the overheads are low.

Our other two implementations (in WiredTiger [51] and RocksDB [50]) demonstrate the generality of our approach. WiredTiger has a substantially different caching architecture than LevelDB, and yet we readily integrated Symbiosis into it with minor code alterations. In doing so, we also discovered a caching bug (acknowledged by the MongoDB team as major); we both fix the bug and show that Symbiosis improves performance. Finally, RocksDB can be configured to avoid the kernel cache; its two-level application-managed caching structure consists of a compressed cache of data read from disk and an uncompressed cache to service queries. We show Symbiosis works well when the application manages both caches directly, again improving performance.

The rest of this paper is structured as follows. We introduce the cache partitioning problem and its significance (§2). Then, we conduct a simulation study of the general two-level cache partitioning problem to guide the design, approximations, and optimizations of Symbiosis (§3). We present Symbiosis’s design and implementation, including its incorporation into LevelDB, WiredTiger, and RocksDB (§4). Finally, we perform an evaluation of our system (§5) using both synthetic and real workloads. We show that our approach improves performance, in some cases by an order of magnitude. We also show the costs of online simulation are not high and various optimizations work well. Overall, we show that Symbiosis is an effective approach to cache-size configuration for modern key-value storage systems.

2 Motivation and Framework

Databases and key-value stores utilize similar caching architectures (Figure 1). Irrespective of underlying data structure organization (log-structure-merge trees [23, 50] or B-trees [51, 61]), these systems use both a custom application-level cache and the underlying file system page cache.

To access a key-value pair, a request first queries an index-like structure, and, if successful, searches for the value in the

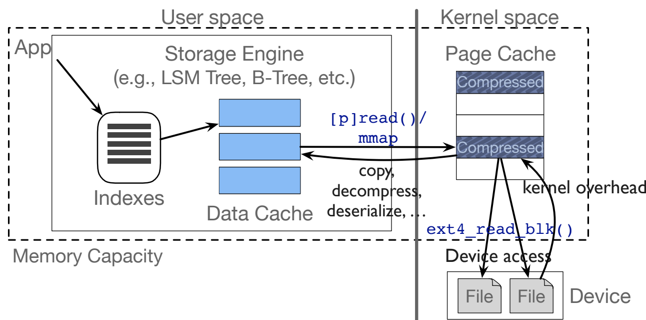


Figure 1: **The Cache Architecture across the Storage Stack**. *Modern applications commonly utilize storage engines (e.g., LevelDB) to manage on-disk data. A storage engine keeps compressed data on disk, and usually has separate index structures and an in-memory buffer for uncompressed data. The arrows depict the common read path.*

user-level *application* cache. If the value is not present in the application cache, a file system read request is issued to fetch the data. This read request may be served by the *kernel page cache*, which holds a compressed version of the data. If the file is not present in the kernel cache, the file system issues necessary I/Os to complete the request, and then caches the (compressed) data. In data-intensive workloads, memory used by the application and kernel caches constitutes a majority of the storage engine's memory usage [14, 30].

Most mainstream storage engines prefer the kernel page cache for buffering on-disk data, to utilize its robust performance under various workloads and to avoid the labor of implementing a sophisticated user-level device-friendly caching and prefetching approach. Thus, we focus our study on this application-kernel cache structure. However, some storage engines can be configured to manage their own second-level cache for compressed on-disk data (e.g., RocksDB). As we will see later, our techniques also work well on this (simpler) user/user configuration.

2.1 The Application-Kernel Cache Structure

We now describe the main properties of two-layer caching. In the first layer, storage engines keep decompressed and deserialized data. These application caches store ready-to-use data to serve requests efficiently.

For example, LevelDB [23], the main storage engine we study, is an LSM-based key-value storage engine with a block-based application cache. Data blocks are variable-sized and not aligned. When a thread inserts an item and overflows the cache, it is responsible for performing evictions using LRU replacement. In contrast, WiredTiger [51], the underlying storage engine of the popular database MongoDB, is a B-Tree-based engine and has a significantly different caching mechanism. Instead of a unified cache structure, WiredTiger constructs an in-memory B-Tree representation and allows each B-Tree node to dynamically allocate memory to cache data. When the total amount of cached data reaches the limit, background threads are initiated to traverse the tree and perform evictions. Each node records last-access recency to approximate LRU replacement.

The second layer of this cache structure is a compressed cache that commonly utilizes the underlying OS kernel's page cache. Storage engines compress on-disk data to reduce device bandwidth and save space on disk; furthermore, by using the kernel page cache, one can leverage years of performance tuning that is present therein.

In Linux, the eviction algorithm is 2Q with a clock algorithm for each queue and involves sophisticated heuristics for promotion, demotion, and size partitioning among the queues. In addition, Linux performs read-ahead to ensure high bandwidth utilization. The current read-ahead approach uses heuristics to determine which pages/when to prefetch (including basing its decisions on the cache presence of pages neighboring the target page), which can significantly affect hit ratio in some scenarios.

To summarize, this two-level cache structure has several important characteristics. First, the application and kernel caches form a two-level caching scheme that shares the same memory quota (i.e., if one cache grows, the other must shrink). The kernel cache often stores compressed data, making it more efficient in terms of memory usage, while the application cache provides lower latency as its data is ready to be used, saving the cost of decompression and kernel crossing. Second, with data compression, the two caches store data in different forms, units, and alignments. One block in the application cache may correspond to several pages in the kernel page cache due to misalignment, which further complicates the management of the two caches and the optimization of overall performance.

2.2 Challenge: Memory Partitioning

Given this two-level caching architecture, a natural question arises: how should memory be allocated between the two caches, in order to maximize performance? To illustrate some of the complexities of this issue, we present the following motivating experiment. Here, we study the performance of different cache configurations in two representative storage engines, LSM-based LevelDB [23] and B-tree-based WiredTiger [51]. We run uniform random workloads with 1 GB of available memory. We use small data sets here to speed our analysis; as we will show later, results are nearly identical when data sets are scaled up.

We compare two extremes: one which devotes all available memory to the application cache, and the other which devotes all memory to the kernel cache. We show how performance varies across two different data set sizes (D_u), 1 GB and 2 GB (uncompressed); the compression ratio is 0.5. Figure 2 presents our results.

We see similar trends from both storage engines. When the data set size is 1 GB (and hence fits, uncompressed, into the application cache), devoting as much memory as possible to the application cache outperforms the kernel-cache by

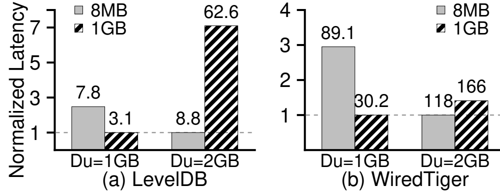


Figure 2: **Storage Engine Performance Varying Data Set Size**. *Each bar depicts one application cache size (8MB or 1GB); each pair of bars shows performance for a given dataset size. The y-axis is the latency normalized to the lowest value; numbers above are absolute latencies (us/op).*

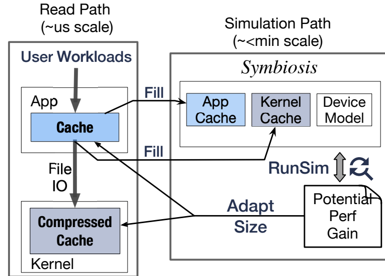


Figure 3: **Overview of Symbiosis**. *This figure shows the main components of Symbiosis and their interactions.*

2.5× to 3×. In contrast, when the data set size is 2 GB (and hence fits compressed into the kernel cache, but is too large for the uncompressed application cache), the kernel cache outperforms the application cache, by up to 7×.

The experiment demonstrates that cache configuration impacts performance significantly; no single configuration performs well across different workloads and settings. A deeper understanding of the performance characteristics of this two-level structure is required; a systematic approach that can coordinate the two caches to maximize performance is needed.

2.3 Cache Coordination with Symbiosis

To address this problem, we propose *Symbiosis*, a system to coordinate application and kernel caches to maximize performance across differing workloads and system configurations. Figure 3 presents an overview of the system architecture. A key element of Symbiosis is an online cache simulator that monitors performance levels given the current application/kernel configuration and determines necessary adaptations to improve performance. The simulator selectively applies *ghost caching* [19] to determine whether a different application cache size would be beneficial; if so, it changes the size of the application cache (and thus implicitly makes more or less memory available for the kernel cache).

Detailed online simulation can be prohibitively slow. Therefore, Symbiosis uses a simplified representation of the actual caching approaches used by real systems. The core challenge thus lies in determining how to abstract the essence of the cache sizing problem and adopt the right level of simplification, aiming for a balance between overhead and accuracy. We show how to strike this balance later (§4).

3 The Cache Partitioning Problem

Through offline simulations, we show the factors that influence how memory should be divided between the application and kernel caches. Our simulations demonstrate that the division of memory between application and kernel caches has a large impact on performance (e.g., up to 9×), and that the best division is highly dependent on a wide variety of factors, some of which are specific to the environment (e.g., application and kernel miss costs) and some of which can vary depending upon workload (e.g., the size of the data set, compression ratio, and application/kernel cache hit rates).

3.1 Influential Factors

We define a number of system and workload parameters that impact the best division of memory.

Memory Cache Sizes: M depicts the total amount of memory that can be used for the application cache (M_a) and kernel cache (M_k); $M_a + M_k = M$. M can represent the total physical memory on a single machine, a containers' resource limit [26, 38, 72], or enforcement by other mechanisms [71, 78]. We arbitrarily fix M to 1 GB in the simulations, since only the relative size of memory to the data size matters, and not its absolute size.

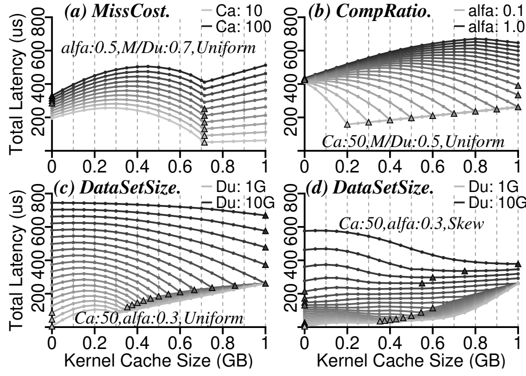
Data Size: The amount of compressed data that is stored on disk by an application is D_c; the corresponding uncompressed data size is D_u. We simulate 1GB $\leq D_u \leq$ 10 GB.

Compression Ratio: (α, $0 < \alpha \leqslant 1$): The ratio of compressed data to decompressed data is α (i.e., $\alpha = \frac{D_c}{D_u}$). α is affected by the compressibility of the data and the specific compression algorithm [73]; for instance, in WiredTiger, we found that compressing a data set of $D_u = 1$ GB using four different compression algorithms (zstd, zlib, snappy, and lz4) takes between 9μs and 204μs and results in compression ratios between 0.36 to 0.51. We simulate values of α between 0.22 (observed in production [13]) and 0.5 (the default for RocksDB's *db_bench* [18]).

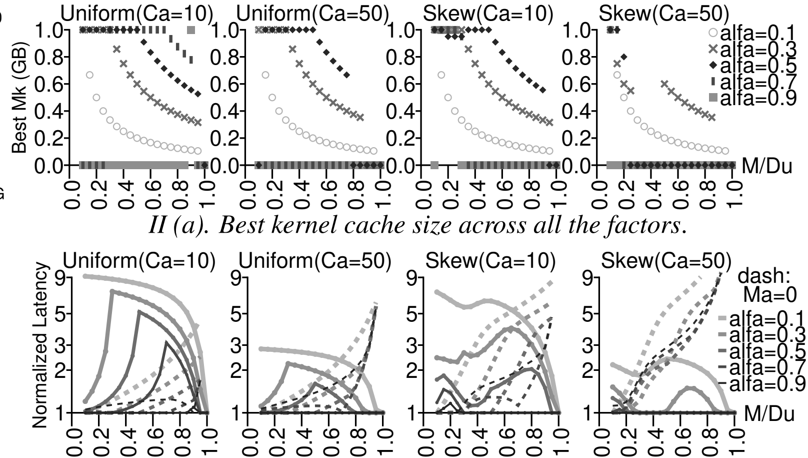
Retaining Data Size: (D_{mem}): We find the notion of a *retaining data size* useful: the size of the resulting data when it is all decompressed from memory. The minimum D_{mem} occurs when all of M is devoted to the uncompressed application cache; that is, $D_{mem}^{min} = M$. The maximum D_{mem} occurs when all of M is devoted to the compressed kernel cache (i.e., $D_{mem}^{max} = \frac{M}{\alpha}$). A higher D_{mem} reduces device accesses.

Hit Rates: The hit rate of the application cache is H_a and the kernel cache is H_k. Hit rates are functions not only of the cache sizes, but also of access patterns and cache replacement policies. We examine uniform random, skewed, and mixed access patterns. Our simulations focus on LRU; note that improvements in replacement policies [9] are complementary to our approach as we aim to better use available memory regardless of the policy.

Miss Cost: Application miss cost is C_a and kernel cache miss cost is C_k. C_a is highly application dependent; empirically, we found C_a varied between 40μs and 250μs de-



I. Performance Varying One Factor. *In each subplot, the title indicates the varied factors across lines; the legend describes parameters of the minimal and maximal value for a factor (the rest is omitted). The triangle indicates the point of the global minima; the bold text depicts the controlled factors.*



II (b). Performance Gain. Two baselines: M_k=0 (solid) and M_a=0 (dashed).
II. Best Configurations. *The title of each subplot means the workload and miss cost. We use $\frac{M}{D_u}$ from 0.1 to 1.0 (x-axis) and two miss costs C_a=10,50.*

Figure 4: **Simulation Results**.

pending on the compression algorithm in WiredTiger and is $< 10\mu s$ in LevelDB; thus, the simulation varies C_a from 10 to 100. The main factor influencing C_k is device performance; we set C_k to $100\mu s$ for common devices. Again, the ratio of miss costs ($\frac{C_a}{C_k}$) matters and not their absolute values.

3.2 Analysis

Our goal is to find the value of M_k that optimizes performance given the other system and workload parameters; our offline simulations do this by sweeping through the full range of valid values of M_k. To quantify the performance of the cache structure, we use average latency: $L_e = (1 - H_a) * (C_a + (1 - H_k) * C_k)$. Generally, as M_k increases, H_k increases, but H_a decreases; thus, the ideal hit rates for H_k and H_a depend on the relative values of C_k and C_a.

3.2.1 Uniform Workload

We begin simulations with a uniform workload as it leads to the most intuitive results. With a uniform workload and LRU replacement, the hit rate of a given cache is simply its size divided by the data size; specifically, $H_a = \frac{M - M_k}{D_u}$ where $0 \leqslant M_k \leqslant M$, and $H_k = \frac{M_k}{\alpha * D_u}$ where $0 \leqslant M_k \leqslant \alpha * D_u$. L_e can be calculated as a quadratic function of M_k with a negative quadratic term coefficient; thus, the two boundary points of the domain ($M_k = 0$ and $\min(M, \alpha * D_u)$) are two local minima, but which of the two is the global minimum depends on all factors, as we illustrate.

Miss Cost (C_a vs. C_k): We begin by showing the best kernel cache size as a function of miss costs. In our two-layer caching architecture, the ratio $\frac{C_a}{C_k}$ determines how much each miss rate contributes to overall performance. While this ratio does not impact the cache configurations of the two local minima, it does influence which is the global minimum.

Figure 4 I(a) shows latency as a function of M_k, varying C_a from 10 to 100 (interval=10) and fixing $D_u = 1.43$ GB (i.e., $\frac{M}{D_u} = 0.7$) and $\alpha = 0.5$. For all values of C_a, the local minima are at $M_k = 0$ and $M_k = \alpha * D_u$, and the global minimum changes from 0 to $\alpha * D_u$ as C_a decreases (i.e., when $C_a < 60$). In general, when $0 < M_k < \alpha * D_u$, L_e is larger than at both extremes because both caches are non-zero and contain duplicates; when M_k grows beyond $\alpha * D_u$, L_e increases because the kernel cache already holds all compressed data. Additional M_k causes more application cache misses. With a higher C_a, the global minimum of M_k is smaller, as application cache misses are penalized more.

Figure 4 II(a) summarizes the best kernel cache size for different parameters, illustrating that different systems and workloads benefit from very different cache configurations, with best values of M_k from 0 to M and all points between. More specifically, the first two subplots show uniform workloads; comparing points across these first two subplots confirms that a higher value of C_a (i.e., $C_a = 50$ vs. $C_a = 10$) makes the best kernel cache size smaller. Figure 4 II(b) shows how much latency is improved when the cache system is configured correctly; specifically, the graphs compare latency with the best cache partition to two reasonable default cache configurations: $M_a = 0$ (dashed lines) and $M_k = 0$ (solid lines). For example, with a smaller C_a, latency can be nine times larger with a poor choice cache configuration (i.e., $M_k = 0$) than with the best choice.

Compression Ratio (α): Figure 4 I(b) shows the impact of α on the best kernel cache size, by varying α from 0.1 to 1 with an interval of 0.05 and setting $D_u = 2$ GB and $C_a = 50$; D_u is set larger than M so that it is not possible to cache all uncompressed data in memory.

Given a lower α (for a fixed D_u), a larger kernel cache tends to be better as it is more efficient with compressed data; with a low α, the kernel cache provides larger D_{mem}, avoiding more device accesses than the application cache. Specifically, with a very low α (i.e., the bottom line with $\alpha = 0.1$), latency drops sharply from $M_k = 0$ to $M_k = \alpha * D_u = 0.2$.

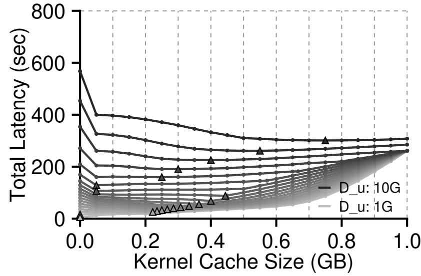


Figure 5: **Simulated performance under a Mixed (Read+Scan) workload**. *The legend describes parameters of the minimal and maximal value for the varying factor, DataSetSize (i.e., D_u). The triangle indicates the point of global minima. (Controlled factors: $C_a = 50, \alpha = 0.2$)*

Generally, while the latency at $M_k = 0$ remains the same, the latency at $M_k = \min(M, \alpha * D_u)$ decreases with smaller values of α; as a result, the global minimum changes from $M_k = 0$ to $M_k = \min(M, \alpha * D_u)$ when $\alpha < 0.65$.

Figure 4 II(a) confirms that larger kernel caches are more beneficial with smaller values of α and Figure 4 II(b) shows that the performance improvement is more dramatic with smaller α; the potential benefit of the kernel cache is high.

Data Size (D_u) **vs. Memory Capacity** (M): Figure 4 I(c) shows the impact of varying D_u from 1 GB to 10 GB (i.e., varying $\frac{M}{D_u}$ from 0.1 to 1.0) while $\alpha = 0.3$ and $C_a = 50$. While the two local minima for M_k (0 and $\min(M, \alpha * D_u)$) follow the studied trends of L_e, we make three specific observations. First, when D_u is very small, the application cache can fit all of the data uncompressed, so all memory should be devoted to the application cache ($M_k = 0$). Second, when D_u is much higher than M (e.g., when $D_u = 10$ GB), the impact of different values of M_k is smaller since most accesses miss both caches. Finally, as D_u grows larger than 2 GB, the global minimum changes from $M_k = 0$ to $M_k = \min(M, \alpha * D_u)$; for these values of D_u, the larger M_k is better because it leads to a larger D_{mem} at the cost of a lower H_a. In summary, the best M_k tends to be 0 for a very large or very small D_u, and $\min(M, \alpha * D_u)$ for a medium D_u.

In Figure 4 II(a), the $\alpha = 0.7$ line in the first graph shows this trend best. As shown in Figure 4 II(b), with a medium D_u, the performance gain over $M_k = 0$ is large and with a small D_u the gain over $M_a = 0$ is generally larger; with a very large D_u, the gain is small as all cache configurations perform similarly.

3.2.2 Non-Uniform Workload

While the hit rates (and thus the best values of M_k) can be precisely calculated for uniformly-random workloads, in practice, most real-world workloads are more complex [13, 17]. We simulate a skewed workload containing a hotspot with locality as suggested by production RocksDB [13] in which 20% of the key space serves 80% of requests. Figure 4 I(d) shows that this skewed workload exhibits a significantly different performance curve from a uniform workload (Figure 4 I(c)). The trend observed for a uniform workload, in which the best M_k grows with increasing D_u, does not hold for skewed workloads and the best M_k becomes highly unpredictable. Generally, for a skewed workload, a larger application cache is preferred since more accesses occur within a smaller hotspot and the same size of application cache provides a higher hit rate; this effect can be roughly viewed as effectively reducing D_u. Figure 4 II(a) shows this preference to the application cache, comparing the right half of graphs to the left half; Figure 4 II(b) confirms that the performance gain over $M_k = 0$ is smaller than for uniform workloads and that over $M_a = 0$ is larger.

Our second non-uniform workload contains a mix of *read* and *scan* operations, as commonly found in real deployments [13, 17]. We use the YCSB benchmark [17] to generate 90% reads and 10% scans with an 80/20 hotspot and a scan length uniformly distributed between 0 and 100 KB. The results in Figure 5 show that the trends are even more irregular: although the best M_k increases with decreasing $\frac{M}{D_u}$ (i.e., increasing D_u), the best M_k decreases significantly when $\frac{M}{D_u}$ decreases from 0.45 to 0.4, and never at the extreme points (i.e., 0 and M) when $\frac{M}{D_u} < 0.9$. In summary, the best cache configuration for a non-uniform workload is more difficult to predict with an offline simulation or model.

3.3 Discussion

Our simulations have shown that the best cache configuration is highly sensitive to factors such as memory capacity, compression ratio, and miss cost, which depend on data and hardware; non-uniform workloads further exacerbate the complexity. The performance gain curves in Figure 4 II(b) show that improvements compared to a default cache configuration can be significant, but that the best kernel cache size varies significantly. Statically determining the best configuration is impractical due to the dynamic nature of workloads, directing us to a runtime adaptive approach. Fortunately, although the amount of gain is difficult to predict, the curves are relatively smooth without abrupt changes, indicating that some inaccuracy in online simulation can be tolerated.

4 Design and Implementation of Symbiosis

We present our design and implementation of Symbiosis, which performs online cache simulation to dynamically and adaptively configure two levels of cache for high performance. The key challenge is to achieve simulation accuracy and configuration coverage while maintaining high performance to minimize the impact on the foreground workload.

4.1 Design

Symbiosis is an add-on module built into a storage engine that automatically adjusts the application cache size (M_a), implicitly changing the kernel cache size (M_k). Figure 6 illustrates how Symbiosis integrates into existing storage engines. Symbiosis contains two main components:

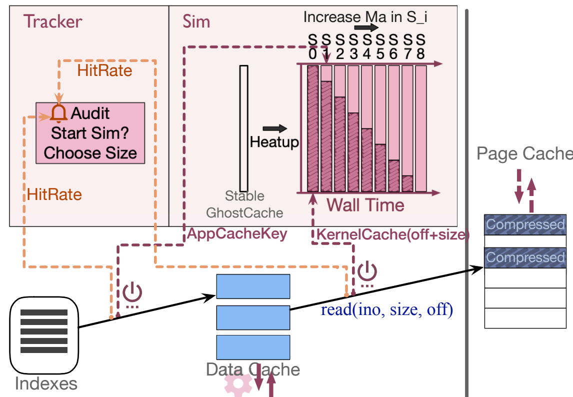


Figure 6: **Design of Symbiosis**. *Symbiosis is directly integrated into a storage engine. The orange dashed lines are the stats collection paths that are always active; the dashed red lines are the paths filling entries into the ghost cache, activated only in* Adapting State *and empty in* Stable State. *The information inside the GhostSim component illustrates how the ghost cache changes across the nine configurations during one simulation round. The size of the application cache (i.e., light red portion of a bar) is increased over time; the dark red portion represents the kernel cache.*

Tracker and *GhostSim*. Tracker continuously audits application and kernel cache accesses to collect performance statistics; Tracker decides when to activate GhostSim to find a better $<M_a, M_k>$ and which specific candidate to adopt. GhostSim uses efficient online cache simulation to predict the performance of candidates.

We design Symbiosis to achieve several goals. First, *low overhead*: incur negligible overhead for the in-memory read path, taking less than a few microseconds if a request hits in the caches. Second, *memory efficient*: minimize memory to reduce interference with the memory-constrained storage engine. Finally, *robust performance*: deliver superior performance in most cases, while guaranteeing baseline performance for arbitrary workloads.

To minimize the overhead of configuration exploration and changes, GhostSim is activated only when necessary. To lower our overhead and memory consumption, we maximize ghost cache reuse with a pipelined simulation of $<M_a, M_k>$ configurations in the order of increasing M_a. To reduce memory consumption and maintain high accuracy, we use sampling specifically tailored to our cache structure, accounting for misalignment and read-ahead in the kernel cache. Finally, to guarantee performance improvements, we apply a policy to guard against (uncommon) inaccurate simulation results.

4.1.1 Auditing by Tracker: Metric and States

Symbiosis alternates between two states: *Stable* and *Adapting*. In the initial stable state, Tracker detects workload changes using the *expected latency*, calculated as $L_e = (1 - H_a) * (C_a + (1 - H_k) * C_k))$. L_e focuses on two major factors: H_a and H_c (and consequently the relative cache sizes) and the relative impact of each type of miss. Specifically, Tracker continuously audits the hit/miss result of each cache and calculates L_e with statically configured miss costs by offline measurement. Tracker periodically compares the current calculated L_e to the initial L_e for this round; if the difference is larger than a fixed threshold (currently 10%), Tracker considers it a workload change and enters the adapting state that starts a simulation round. Thus, GhostSim is activated only when necessary.

4.1.2 Simulating with GhostSim: Lifetime of a Round

The basic idea of the adapting state is to systematically generate several $<M_a, M_k>$ candidates, run simulations to predict their L_e's, and determine if the best of them has sufficient performance gain to be applied to the real system. GhostSim is responsible for efficiently predicting the performance of different cache configurations for the current workload. To simulate live workloads and predict their expected latency, GhostSim maintains a *ghost cache* [19,22,53,75], filled with the same indices as in the embedded storage engine, but without the actual data. To minimize memory consumption and performance overhead, GhostSim simulates only one instance of ghost cache at a time, adopting a pipelined simulation of candidates in the order of increasing M_a to maximize ghost cache reuse. After collecting the L_e of each candidate $<M_a, M_k>$ through simulation, Tracker derives the potential gain of the best candidate configuration and applies it to the real system if the gain surpasses a certain threshold. The ghost cache entries are then discarded to save memory. Symbiosis waits for the real caches to warm up and generate a stable initial L_e as the reference point in the next period.

We strictly bound the ghost cache's space and time overhead with a collection of techniques (described below), as a naive full simulation incurs unacceptable memory consumption ($> 5\%$) and performance overhead ($> 30\%$).

4.2 GhostSim Optimization Techniques

We introduce four techniques to achieve sufficient simulation accuracy, memory efficiency, performance, and robustness; overall, we identify and solve new challenges for sampled ghost cache simulation raised by the unique interaction pattern of the two-level cache structure. First, we reset to a cache configuration during simulation that will perform reasonably for the current workload; second, we simulate a pipelined sequence of candidate configurations to achieve high coverage and efficiency; third, we use sampling to achieve accurate simulation with reduced memory; fourth, we guard against (uncommon) flawed simulation results that could occur due to not modeling all kernel caching features.

4.2.1 Initialization: Reset Policy

During *Adapting State*, GhostSim must use a cache configuration that performs reasonably for the live foreground workload; GhostSim either continues using the current cache configuration, or if L_e has increased (likely from an increase in D_u), it resets to the minimal default M_a used by the original storage engine (which increases D_{mem}). We show the benefits of this reset policy in Section §5.2.4.

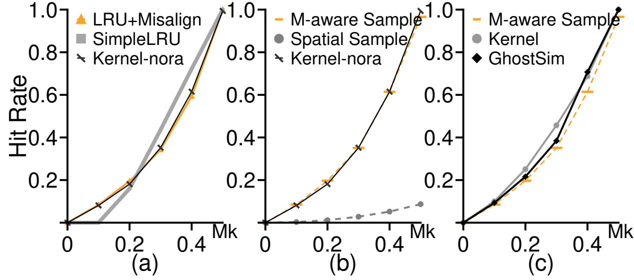


Figure 7: **KernelCache Simulation and Sampling**. *Kernel-nora and Kernel are the kernel cache implementations with and without read-ahead, respectively.*

4.2.2 Incremental reuse of a single Ghost Cache

We extend the idea of storing cache access metadata with a ghost cache [19, 22, 53, 75] to efficiently handle two-levels of caches while minimizing the memory footprint. Multiple first-level cache sizes can be simulated simultaneously with only the amount of memory required for the largest cache if the first-level cache follows the stack property [48] (e.g., LRU). However, the second-level cache sees different access patterns depending on the size of the first level, and thus has different contents when sized differently. Thus, simultaneous simulations of all second-level size candidates within one ghost cache instance is infeasible.

To efficiently simulate memory configurations with ghost caches, Figure 6 illustrates our choices of $<M_a, M_k>$ candidates. Our simulation results (Figure 4 II(a)) indicated that the best memory configuration could be anywhere within the search space; therefore, GhostSim forms the candidate set by dividing the search space into a fixed number of equal ranges (currently 8) without skipping candidates or stopping early; this provides relatively high coverage of the search space with reasonable convergence time. Since warming up each candidate ghost cache is a significant source of overhead, Symbiosis simulates each in the order of increasing M_a to maximize the reuse of ghost cache contents. Specifically, we keep the application ghost cache at its full size and simulate different M_a's using the stack property, so that when M_a is increased for the next candidate, the contents of the increased portion are already known. A short warm up for the kernel ghost cache after M_k is decreased is required to let its contents approach those of the next candidate's configuration.

4.2.3 Sampling with Misalignment and Read-ahead

Even with reuse, the memory consumed by a ghost cache is significant (e.g., 50 MB for 1 GB data). To reduce memory consumption, we incorporate a key-space sampling technique by hashing the indices so that one slot represents several keys [67, 68]. A sample ratio (R) of 0.01-0.001 minimizes memory usage while preserving accuracy.

Approximating H_k with sampling poses new challenges. An important difference between Symbiosis and other two-layer cache structures is that the kernel caches at the page level while the application caches in application-defined blocks that misalign with pages; as a result, the independent reference model [2] does not hold, as each request may access different targets in each layer and multiple contiguous targets in the kernel cache. Moreover, read-ahead strongly affects H_k, but a full simulation would be too costly.

We introduce different hashing approaches that accurately model these real-system effects. Figure 7 shows the hit rate curves for various kernel cache implementations and sampling approaches. Figure 7(a) shows a SimpleLRU simulator that caches in the unit of blocks instead of pages and thus does not take misalignment into account, deviates significantly from a kernel implementation that has read-ahead disabled (Kernel-nora). The LRU+Misalign simulation, which caches in the unit of pages and accounts for misalignment just as the kernel does, approximates the Kernel-nora line well. However, Figure 7(b) shows that spatial sampling ($R = \frac{1}{2}$) is not effective in the presence of misalignment, deviating from the Kernel-nora line. With misalignment, accessing a block across pages will read both pages into the cache, hitting neighboring blocks; spatial sampling's hashing scheme loses locality and cannot capture such behavior. We introduce *misalignment-aware sampling* that groups contiguous G application blocks before hashing to preserve locality; the M-aware Sampling line ($R = \frac{1}{2}$ and $G = 32$) approximates the Kernel-nora line well. Finally, to compensate for read-ahead, we adopt a heuristic that slightly increases the size of our modeled kernel cache. Figure 7(c) shows that this final version (GhostSim) approximates the Kernel better than M-aware Sampling.

Our sampling method produces similar hit rate curves with $R \geqslant \frac{1}{256}$; we choose $R = \frac{1}{64}$ due to the acceptable variance and sufficiency to realize a low-overhead online simulation. We confirm that our method broadly works well.

4.2.4 Guard against Unmodeled Cases and Fall Back

Although we have modeled misalignment between caches, GhostSim may be inaccurate in some workloads due to unmodeled kernel features such as read-ahead. Thus, Symbiosis only performs cache size adjustment if the predicted result improves latency by a threshold amount; we do not adapt away from settings that already works well. To understand why this approach is robust, consider a workload that performs strided access of one key per page. The kernel cache sees a linear access, triggers read-ahead, and thus achieves a high H_k, while GhostSim without read-ahead produces a low H_k. However, Symbiosis observes that the predicted L_e for all the candidate cache sizes is larger than the measured current L_e, and therefore rejects all simulation results.

4.2.5 Limitation and Discussion

We assume that workloads change infrequently. If the workload changes before a simulation round ends, Symbiosis detects the change, discards the current results, and starts over. If the workload changes repeatedly during simulation, Symbiosis stops the simulation as it is unable to finish and yield

benefits. In our experimental environment, Symbiosis takes at most 45 seconds to detect and simulate new workloads.

Symbiosis generally offers larger and more robust benefits to existing storage engines in read-heavy workloads, which are observed as dominant in various studies [13, 17]. The idea of simulation-based cache size adaption can work with write-heavy workloads, yet will require additional research to realize in robust form. For example, LSM-based engines often schedule asynchronous background compaction in the write path; thus, speed differences in the foreground workload caused by different cache size configurations can lead to varying tree structures and thus different cache access traces. Further, write performance itself is less stable than read performance [8], which is more challenging for prediction.

4.3 Multiple Implementations

We have integrated Symbiosis into three different storage engines: LevelDB [23], WiredTiger [51], and RocksDB [50]. Modifying LevelDB to leverage Symbiosis required adding fewer than 1000 LoC to the 30000-LoC codebase. First, the required keys for the ghost cache are collected during the original processing of each request. Second, hit/miss statistics are recorded when accessing the application cache and inferred from timing when accessing the kernel cache. Third, LevelDB's `LRUCache` is modified to build the ghost cache utilizing the stack property, greatly reducing the amount of new code. Finally, a generic interface is added to the application cache to dynamically resize it to M_a and allow the kernel cache to automatically use the rest of the memory ($M - M_a$).

We have also ported Symbiosis to WiredTiger and RocksDB to demonstrate its generalizability. Despite the fact that WiredTiger's B-Tree-based engine has a completely different caching mechanism than LevelDB, the modifications required are similar to the four outlined above; the basic port added fewer than 100 LoC to WiredTiger and Symbiosis. Interestingly, as part of this porting process, we uncovered a bug in WiredTiger's cache eviction mechanism. Despite its claimed LRU-like behavior, the bug makes it evict data regardless of recency and its cache performance becomes extremely poor and unpredictable. This bug has been reported to MongoDB which recognized it as a major bug; we have added a workaround to restore the intended LRU policy, which significantly improves performance and enables Symbiosis to correctly simulate its cache behavior.

RocksDB is based on LevelDB and has a similar caching mechanism. To study Symbiosis's capability to handle an application-managed compressed data cache, we enable RocksDB's option to use its built-in compressed data cache and direct I/O. Whenever the application cache size is changed, we explicitly set the size of the compressed data cache to be all memory not used by the application cache (i.e., $M - Ma$). Due to RocksDB's similarity to LevelDB, the port required minimal effort.

Table 1: **Factors for Static Workload**. *Access patterns are generated by YCSB [17]. Zipfian has scattered hotspots over the key range to avoid space locality. Hotspot{30,20,10} means that 70%, 80%, and 90% of requests access 30%, 20%, and 10% keys in a contiguous range.*

Factors		Presented Space
Workloads	Data Set Size (GB)	5, 2.5, 1.67, 1.25, 1 ($M : D_u$= 0.2, 0.4, 0.6, 0.8, 1)
	Access Pattern	uniform, zipfian, hotspot{30,20,10}
Software	Compression Lib	snappy (default), zstd
	Storage Engine	LevelDB (default), RocksDB, WiredTiger
Hardware	CPU Freq.	HW1: Xeon 5128R (2.9 GHZ) HW2 [57]: Xeon D-1548 (2.0 GHz)
	Device Latency	HW1: OptaneSSD 900P ($\sim 10\mu s$) HW2: Toshiba NVMe flash ($\sim 70\mu s$)

5 Evaluation

We evaluate Symbiosis to answer the follow questions: (1) How much better does Symbiosis perform than reasonable static cache size configurations ($<M_a, M_k>$) for different data set sizes (D_u), compression ratios (α), miss costs (C_a and C_k), and access patterns for different storage engines such as LevelDB, WiredTiger, and RocksDB? (2) How quickly does Symbiosis react to workload changes and how much overhead does Symbiosis incur for simulation and changing cache sizes? (3) How well does Symbiosis handle real-world workloads?

Setup. We use HW1 in Table 1 unless otherwise noted; the available memory M is fixed at 1 GB by cgroup. We evaluate Symbiosis by comparing it with two static configurations: M_a = 8 MB (LevelDB's default) and M_a =1 GB ($M_k \approx 0$), referred to as $Static_{M_a=8MB}$ and $Static_{M_a=1GB}$, respectively.

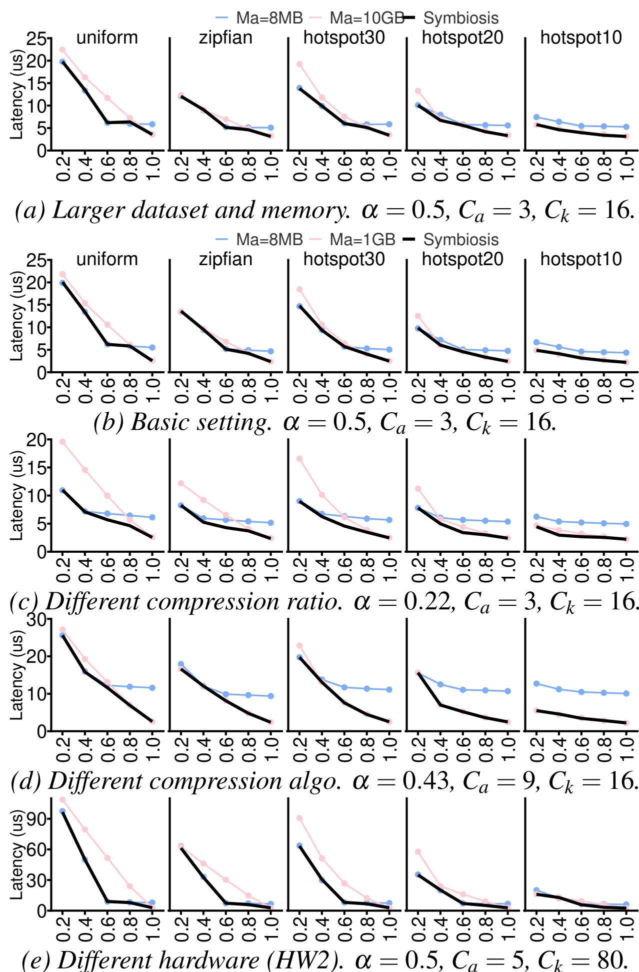
5.1 Static Workloads

We first evaluate Symbiosis under various static workloads, demonstrating that Symbiosis finds a better $<M_a, M_k>$ for different data set sizes (D_u), compression ratios (α), miss costs (C_a and C_k), and access patterns. Table 1 shows the full range of factors. To vary α, C_a, and C_k, we use a secondary compression library (*zstd*) and hardware (HW2). We also evaluate its performance in WiredTiger and RocksDB to demonstrate its generalizability to different storage engines.

5.1.1 LevelDB Performance

Figure 8 compares the performance for LevelDB with Symbiosis to the two static baselines as a function of $\frac{M}{D_u}$ for five access patterns on five different settings.

Large datasets and memory (a): To evaluate Symbiosis in the context of modern data center machines with large amounts of memory, we begin with $M = 10GB$ and a range of large data sets (D_u=50, 25, 16.7, 12.5, 10 GB); we use the basic setting of HW1 and LevelDB's default compression ($\alpha = 0.5$). In all cases, Symbiosis matches the performance of the better baseline. $Static_{M_a=8MB}$ tends to perform better



(a) *Larger dataset and memory.* $\alpha = 0.5$, $C_a = 3$, $C_k = 16$.

(b) *Basic setting.* $\alpha = 0.5$, $C_a = 3$, $C_k = 16$.

(c) *Different compression ratio.* $\alpha = 0.22$, $C_a = 3$, $C_k = 16$.

(d) *Different compression algo.* $\alpha = 0.43$, $C_a = 9$, $C_k = 16$.

(e) *Different hardware (HW2).* $\alpha = 0.5$, $C_a = 5$, $C_k = 80$.

Figure 8: **Performance under Static Workloads**. *X-axis is $\frac{M}{D_u}$. Ma=8MB means $Static_{M_a=8MB}$, similarly for Ma=1GB.*

when the data set is very large, and $Static_{M_a=1GB}$ when the data set size is small; the only exception is hotspot10, where the highly skewed accesses to the small hotspot should always reside in the application cache ($Static_{M_a=1GB}$). Again, Symbiosis dynamically sizes the two caches to obtain the best observed performance.

Basic Setting (b): The setting is the same as (a), except to reduce the running time of our experiments, we use 1/10-th the data set sizes and $M = 1GB$. As desired, the full range of results are extremely similar to that of (a); thus, for efficiency, we use the smaller data set sizes and $M = 1GB$ in the remainder of our experiments.

Different Compression Ratio (c): We change the compression ratio from $\alpha = 0.5$ in (b) to 0.22 in (c). With a smaller α, the performance gap between the two baselines increases, as noted in our offline simulations (§3). Thus, with better compression, Symbiosis achieves a larger performance increase over the worse baseline (commonly $> 1.2\times$) and some improvement over the best baseline (11.1% on average), especially when $M : D_u$ is within $[0.4, 0.8]$.

Different Compression Algorithm (d): We change the compression algorithm to alter α from 0.5 to 0.43 and C_a from 3 to 9. Now, $Static_{M_a=1GB}$ usually performs better than $Static_{M_a=8MB}$ because $\frac{C_a}{C_k}$ is large (0.56) and $Static_{M_a=8MB}$ incurs the cost of the higher C_a. Symbiosis again always matches the performance of the better baseline, properly devoting most space to M_a, while correctly identifying the exceptions (e.g., the leftmost points in uniform and hotspot30).

Different hardware platform (e): We switch to HW2 so that device access is far slower than decompression ($\frac{C_a}{C_k} = 0.0625$). Now, $Static_{M_a=8MB}$ usually performs better than $Static_{M_a=1GB}$ because it avoids costly disk accesses, except for the hotspot10 workload where the cost of frequent application cache misses on the hotspot outweighs the benefit of reduced disk accesses. In several cases (e.g., $\frac{M}{D_u} = 0.8$), Symbiosis performs significantly better than both baselines by properly balancing application cache misses and disk accesses, with an average gain of 6.9% over the better baseline.

Summary: In our LevelDB experiments, Symbiosis achieves as high of performance as the better baseline and outperforms the other baseline by up to 5.77×. In some cases, Symbiosis performs significantly better than both baselines (up to 1.32×), demonstrating the benefit of a fully flexible configuration of $<M_a, M_k>$.

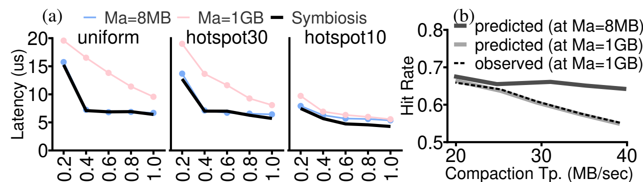
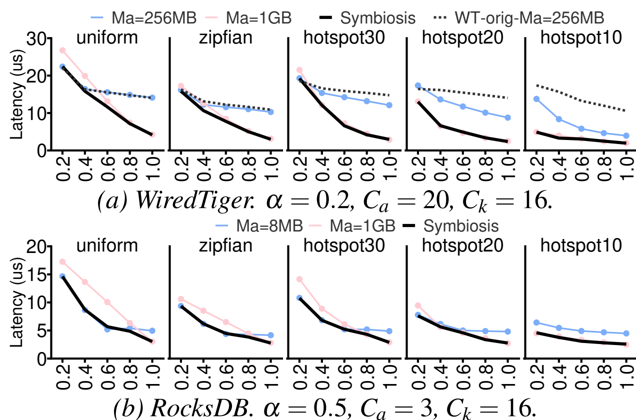


Figure 9: **Static Workload with 20% Overwrites**. *(a) X-axis is $\frac{M}{D_u}$. Ma=8MB means $Static_{M_a=8MB}$, similarly for Ma=1GB. $\alpha = 0.22$, $C_a = 3$, $C_k = 16$. (b) shows the predicted application cache hit ratio of the $Ma = 1GB$ configuration using cache traces from configuration $Ma = 8MB$ and $Ma = 1GB$, and the observed hit ratio when $Ma = 1GB$, under different compaction rates. The workload is uniform with 20% overwrite and $M = D_u$.*

5.1.2 Workload with Writes in LevelDB

During simulations, Symbiosis uses cache access traces from the real system with a certain cache configuration, which deviates from the true cache access traces for other cache configurations when compaction exists. Figure 9(b) shows that Symbiosis's prediction is affected by such deviations under a large compaction rate. By limiting the compaction rate, the inaccuracy can be significantly reduced.

Figure 9(a) shows Symbiosis's performance with 20% overwrites. Compared to its read-only counterpart (Figure 8(c)), $Static_{M_a=1GB}$ performs worse than $Static_{M_a=8MB}$ even when $\frac{M}{D_u} = 1$ due to the immutable nature of LSM-tree that causes duplication with overwrites and makes the actual database size larger. Similarly, Symbiosis offers lower benefits, but still outperforms $Static_{M_a=8MB}$ when the workload is very skewed and D_u is small.



(a) WiredTiger. $\alpha = 0.2$, $C_a = 20$, $C_k = 16$.

(b) RocksDB. $\alpha = 0.5$, $C_a = 3$, $C_k = 16$.

Figure 10: **WiredTiger and RocksDB (Static Workload)**. *X-axis is* $\frac{M}{D_u}$. *Ma=8MB means* $Static_{M_a=8MB}$*, similarly for Ma=1GB. In (a), WT-orig-Ma=256MB is the original WiredTiger, while Ma=256MB, Ma=1GB, and Symbiosis uses our modified WiredTiger with LRU-like eviction policy.*

5.1.3 WiredTiger Performance

Figure 10(a) shows the performance benefits of incorporating Symbiosis into WiredTiger. As mentioned in §4.3, we began by modifying WiredTiger to correctly implement its claimed LRU-like behavior for its application cache; our modified version performs the same or better than the original version (*WT-orig Ma=256MB*) for all static workloads and is used in our baselines (*Ma=256MB* and *Ma=1GB*). WiredTiger has a significantly larger application cache miss penalty ($\frac{C_a}{C_k} = 1.25$) than LevelDB, so even with a very small compression ratio ($\alpha = 0.2$), the baseline with a larger application cache (*Ma=1GB*) performs better than the other baseline for almost all workloads. Since WiredTiger's performance drops significantly when its cache size is less than its 256 MB default, Symbiosis searches for application cache sizes between 256 MB and 1 GB and outperforms or matches the better baseline, showing its capability on a completely different storage engine.

5.1.4 RocksDB Performance

Figure 10(b) shows the performance improvement when RocksDB uses Symbiosis to manage the sizes of its own decompressed and the compressed data cache. Making Symbiosis work with high accuracy is easier in this setting since we do not need to approximate complex kernel cache behavior. These results show a similar trend to that in Figure 8(a) where Symbiosis outperforms or matches the performance of the better baseline, demonstrating its capability to handle application-managed compressed data caches.

5.2 Dynamic Workloads

We demonstrate that Symbiosis adapts to workload changes with a reasonable convergence time and negligible overhead.

5.2.1 Example: LevelDB Behavior over Time

We begin by illustrating how Symbiosis within LevelDB behaves over time for a dynamic workload. Figure 11 presents

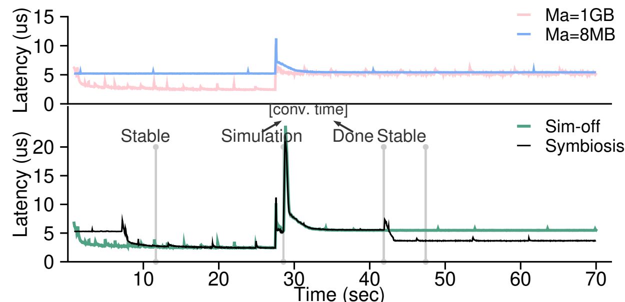


Figure 11: **Timeline of Latency under a Dynamic Workload (hotspot20:1.0-2.0)**. *The workload changes are aligned at* $\sim 26sec$*, and we label state transfer of Symbiosis by the gray vertical lines. Sim-off means we turn off the simulation and shows the effect of only resetting the application cache size to default; its steady performance is the same as* $Static_{M_a=1GB}$ *before the change, and the same as* $Static_{M_a=8MB}$ *afterwards. (*$\alpha = 0.22$*)*

the performance of Symbiosis (the bottom) and the two baselines (the top) for a workload with two phases; the access pattern in both phases is hotspot20 and $\alpha = 0.22$, but D_u varies from 1 GB to 2 GB.

The $Static_{M_a=8MB}$ baseline quickly obtains stable (but relatively poor) performance in the first phase, since the kernel cache can hold all the compressed data. When D_u increases, the latency increases while the kernel cache is warmed with the larger data set, but eventually returns to its previous performance since the kernel cache can still hold all compressed data ($M_k \approx M > \alpha * D_u$ and $H_k = 1$).

The $Static_{M_a=1GB}$ baseline takes longer to warm the application cache in the first phase, but then achieves better performance since the application cache can hold all the decompressed data. When D_u increases, the latency increases because the application cache cannot contain all the data ($M_a < D_u$) and disk accesses are necessary.

Symbiosis is able to obtain as good of performance as $Static_{M_a=1GB}$ in the first phase and better than both in the second. Symbiosis starts with a default value for M_a = 8 MB while simulating cache configurations for $\sim 5sec$; after determining that $M_a = M$ delivers the best performance, it increases the application cache and matches the performance of $Static_{M_a=1GB}$ after the application cache is warmed at $\sim$ $12sec$. After Symbiosis detects the significant increase in L_e at $\sim 28sec$, Symbiosis defaults back to M_a = 8 MB and re-starts the simulations; the large initial overhead is due primarily to warming up the kernel cache (as shown by the Sim-off line which undergoes the same changes in cache configurations without simulation). Once the kernel cache is warmed, the simulation itself incurs negligible overhead (compared to $Static_{M_a=8MB}$) and finishes at $\sim 42sec$, at which point Symbiosis changes to $M_a = 0.5M$, warms up the cache $\sim$ 2 seconds, and then achieves the lowest latency.

5.2.2 Performance Gain and Dynamic Adaptation

To quantify the benefits, convergence time, and resulting cache configurations for a wide range of workloads with two

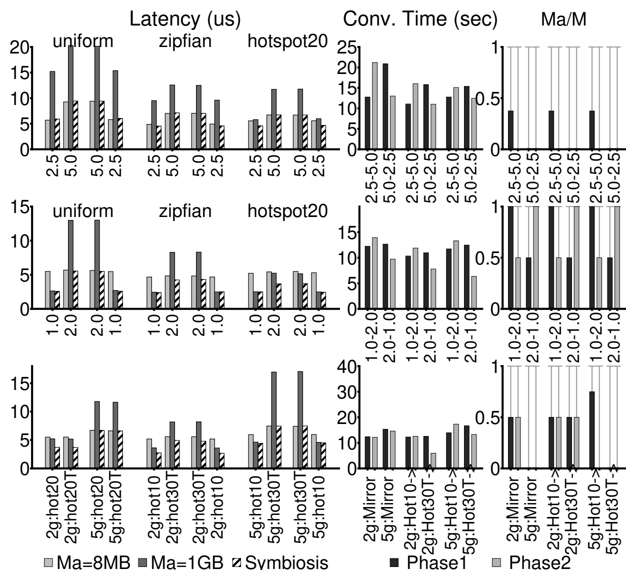


Figure 12: **Performance under Dynamic Workloads** ($\alpha = 0.22$). *In the Latency subplot, each group has three bars: $Static_{M_a=8MB}$, $Static_{M_a=1GB}$ and Symbiosis. Each adjacent bar group represents one workload1→workload2 change and the next group reverses the workloads. The first two rows contains 12 workloads where D_u varies (shown in the x-axis labels). The third row contains 2 workloads varying hotspot positions and 4 varying hotness and hotspot positions, each with a fixed D_u. For instance, 2g:Hot20 means a hotspot20 workload with $D_u = 2$ GB and 2g:Hot20-T mirrors the hotspot to the tail. 2g:Hot20→2g:Hot20-T is summarized as 2g:Mirror (hotspot change). The Conv. Time and Ma/M subplots only show the behaviors of Symbiosis.*

phases, we construct a suite of 18 experiments varying D_u, access patterns, and α (0.22 and 0.5). We present the results with $\alpha = 0.22$ in Figure 12 ($\alpha = 0.5$ omitted for brevity) but consider both αs when discussing extremes and averages.

We use the example above to explain the metrics in Figure 12, which corresponds to hotspot20:1g→2g (the fifth bar group in the second row). Adjacent bars in the figure represent the two phases in an experiment. Latency is reported when performance is stable (e.g., in the example workload, latency is about 2.5μs for Symbiosis and $Static_{M_a=1GB}$ for the first phase, and 5μs for $Static_{M_a=8MB}$; it is about 3.7μs for Symbiosis and 5μs for $Static_{M_a=8MB}$ and $Static_{M_a=1GB}$ in the second phase). Convergence time represents the time to finish simulation (e.g., $\sim$ 12 and 13 seconds for phase 1 and 2, respectively, shown by the time between the bars labeled as Simulation and Done in Figure 11). Finally, the M_a/M subplot shows the best application cache size found by Symbiosis (e.g., 1 and 0.5 for the example workload).

Figure 12 shows that Symbiosis delivers good latency in all cases, at least as good as the best baseline and sometimes better, with an average gain of 24% over $Static_{M_a=8MB}$, 42% over $Static_{M_a=1GB}$, and a best case of 42% over the better of the two (i.e., hotspot20:1.0→2.0). The average convergence time is 15.4 seconds with a worst case of 40 seconds; generally, more convergence time is required for larger D_u and D_c, and for less skewed workloads. During simulation, the worst overhead of Symbiosis is 15.1%, but this contains two portions: the larger is the overhead of possibly resetting M_a to the default and warming up the kernel cache; the smaller is the actual simulation overhead, which averages only 0.9% with a worst case of 3.4%. Finally, Symbiosis chooses different M_a values, typically scaling up M_a with a decrease in D_u and increase in skewness (and vice versa).

Adapting the size online and potential latency spike symptoms raises concerns of tail latency. As shown in Table 2, Symbiosis incurs reasonable tail latency overhead, with a 8.6% higher median p-95 latency and a 15.3% higher median p-99 latency compared to $Static_{M_a=8MB}$. Out of the 18 cases, 13 have less than 25% overhead for p-99 latency. The highest p-99 latency overhead is 52% in uniform:1g→2g. Extra device accesses due to cache size change cause the higher tail latency. Tail latency would be minimally impacted in workloads with a longer steady state or more device accesses.

Table 2: **Tail Latency**. *Overhead is the comparison to $Static_{M_a=8MB}$. ($\alpha = 0.22$)*

	p-95 Latency Median	p-95 Latency Max	p-99 Latency Median	p-99 Latency Max
Overhead (%)	8.6	14.5	15.3	52.0
Case	-	zipfian:1g→2g	-	uniform:1g→2g

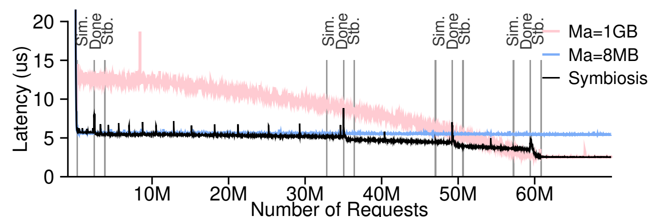


Figure 13: **Timeline of Latency under a Dynamic Workload with Gradual Change**. *The workload is uniform with $D_u = 2$ GB in the first 10M operations, $D_u = 1$ GB in the last 10M operations, and a uniform gradual change during the 50M operations in between. ($\alpha = 0.22$)*

5.2.3 Gradual Change

We show that Symbiosis also performs well in workloads with more gradual changes (Figure 13). During the workload, $Static_{M_a=8MB}$ holds all the data in the kernel cache; $Static_{M_a=1GB}$ cannot hold all the data in the application cache when $D_u = 2$ GB and performs worse, but then benefits from the shrink of D_u and finally eliminates device access when $D_u = 1$ GB and performs better than $Static_{M_a=8MB}$.

Symbiosis matches the performance of $Static_{M_a=8MB}$ at the beginning. Three simulations are triggered when the difference of L_e reaches the threshold for workload change detection, M_a is gradually increased according to the workload when simulations occur, and the latency drops along with the shrink of D_u. Finally, $M_a = M$ is chosen when D_u approaches 1 GB and the performance of $Static_{M_a=1GB}$ is matched.

A gradual change of L_e is necessary for Symbiosis to match the change speed of workload. For workloads with

Figure 14: **Overhead during Simulation** ($\alpha = 0.22$). *The workloads are in the same order as in Figure 12. The bars are the overhead with the reset policy; dashed ones indicate no actual M_a change. Numbers in gray background are the overhead percentages without the reset policy.*

faster changes beyond Symbiosis's threshold during simulation, simulations are halted until the workload stabilizes.

5.2.4 Effect of Optimization Techniques

We quantify the benefits of our techniques by comparing to a simplified version without the corresponding technique.

Reset Policy: The reset policy (§4.2.1) aims for a cache size that performs reasonably while simulating, despite an arbitrary new workload. The overhead of Symbiosis compared to the $Static_{M_a=8MB}$ baseline during simulation is shown in Figure 14; large negative values occur when Symbiosis does not reset M_a to default due to a decrease in L_e and thus Symbiosis performs better than the baseline (e.g., the uniform:2g→1g experiment). As shown by the overhead numbers in gray background in Figure 14, Symbiosis without the reset policy performs poorly in many cases (e.g., up to 100×); therefore, the reset policy is better on average and beneficial for more stable performance.

Sampling: Sampling is essential for low overheads. The first and third row in Table 3 shows the memory consumption and operation overhead of Symbiosis with and without sampling. Without sampling, simulation consumes 51 MB of memory and adds 42% of overhead to every operation. Sampling significantly reduces the costs, consuming only 460 KB of memory and incurring only $\sim 90ns$ per operation. Furthermore, sampling only adds the overhead over the 16.7 second simulation round – a negligible duration.

Incremental reuse of ghost cache: By comparing rows two and three in Table 3, we see that incremental reuse reduces both memory and time overhead by $> 3\times$, but at the cost of a longer convergence time, compared to a design that simply uses one ghost cache instance for each candidate $<M_a, M_k>$. Thus, the incremental reuse design has the lowest impact on foreground workload and is most suitable.

5.3 Real World Workloads

We conclude by demonstrating that Symbiosis handles complex and realistic workloads: performance is robust since only a size change that is predicted to sufficiently improve performance is adopted.

Two workloads generated from RocksDB's *mix_graph* benchmark [13] are used, the first with the supplied parameters in the last example in paper [13], and the second mimicking an interesting two hot key-range symptom in the paper, observed by Meta's ZippyDB Get workload. The benchmark models key-space localities and closely approaches real workloads in terms of storage I/O statistics.

Table 3: **Space and Time Overhead and Convergence Time of Various Simulation Settings**. *Operation overhead compares to baseline LevelDB. Sample rate is $\frac{1}{64}$.*

Case	Memory Overhead (MB)	Operation Overhead (us/op)	Conv. Time (s)
Reuse & No Sampling	51	2.8 (42%)	22.9
No Reuse & Sampling	1.5	0.32 (4.8%)	7.35
Reuse & Sampling	0.46	0.09 (1.3%)	16.7

Figure 15 shows the performance of LevelDB on four consecutive traces based on the two workloads. $Static_{M_a=8MB}$ maintains relatively constant performance through the four phases with $H_k \approx 1$, as the kernel cache holds most of the compressed data across all phases. $Static_{M_a=1GB}$ outperforms $Static_{M_a=8MB}$ in the first and the second phase because the workload is very skewed (over 70% of requests access 1/30 of the data), and the gain of hitting in the application cache for most accesses outweighs the additional disk accesses for the data that does not fit; however, in the third and fourth phases, $Static_{M_a=1GB}$ performs worse than $Static_{M_a=8MB}$ as the workload becomes less skewed, with 80% of requests accessing 40% of the data, lowering H_a.

Symbiosis finds a $<M_a, M_k>$ as good as (and often better than) the better static configuration in every phase of the complex production workload. To illustrate why Symbiosis is robust, the small bar charts show the predicted L_e of $<M_a, M_k>$ candidates from $M_a \approx 0$ to $M_a = M$ and the real L_e (gray line) during each simulation. For each simulation, Symbiosis resets $M_a = 8$ MB. In the first three phases, the best candidate is $M_a = \frac{3}{8}M$ and its L_e is much lower than the real L_e, so Symbiosis applies it to the real system and outperforms both two baselines. In the last phase, the best candidate is $M_a = 8$ MB which is the default value that Symbiosis currently takes, so it keeps the default M_a and matches the performance of the better baseline $Static_{M_a=8MB}$.

6 Related Work

Dynamic Cache Adaptation: As caching performance hinges on workload access pattern, prior work has explored how to dynamically adapt various aspects of cache management. Our work, sharing a similar motivation to effectively adapt to online workload changes, benefits from relevant innovations and operates within a more complex application-kernel cache structure.

In the scenario of a single-level cache where no cooperation is explicitly introduced, such efforts centered around dynamic replacement policies [5, 58, 69], cache allocation and partitioning [20, 28, 36, 39, 49, 54, 60, 64, 65, 82], and online cache performance approximation [37, 46, 59, 67, 68, 74]. For instance, SOPA [69] simulates different cache replacement policies to dynamically decide the best policy. ACME [5] simultaneously runs multiple cache replacement policies and updates their weights by the instant effectiveness. Recently, machine learning techniques were also explored [58].

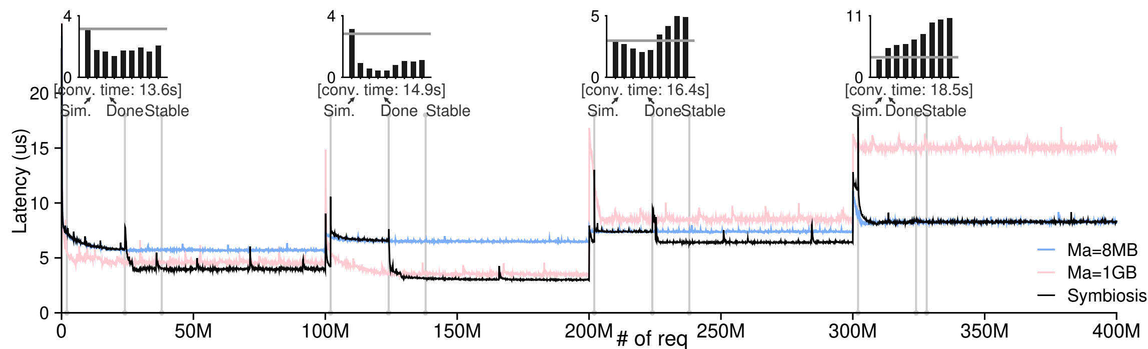


Figure 15: **Request Latency versus the Request Sequence**. *The 4 phases are composed of 2 workloads generated from RocksDB's mix_graph benchmark. Two versions of the first workload exhibit a decrease in D_u, with $Key_{max} = 50M$ and $D_u = 5$ GB in phase 1 and $Key_{max} = 25M$ and $D_u = 2.5$ GB in phase 2. Similarly, two versions of the second workload exhibit a increase in D_u s (phase 3: $D_u = 2.5$ GB and phase 4: $D_u = 5$ GB). The four small bar charts around the top illustrates the decision of Tracker; each chart is a simulation round. Each bar represents one simulated cache size setting ($S_{\{0,\ldots,8\}}$ from $M_a = 8$ MB to $M_a = 1$ GB), y-axis is the L_e (expected latency), and the gray horizontal line shows the real system L_e at the time of simulation end. Tracker adopts the first three size changes, but rejects the last one; all four are good decisions. ($\alpha = 0.22$)*

Caching strategies designed for the properties of a given layer are necessary, such as for flash endurance [16, 27, 29, 53]. Our work, instead, considers compression, as it is widely-used in modern key-value storage engines. Recent research also incorporates compression in storage systems [43, 47, 77, 81], underscoring its importance.

Hierarchical Cache Management: Earlier works have distilled and tackled several major problems introduced by hierarchical cache management [79]: weak temporal locality in the second layer [83] due to the first layer's filtering effect, duplication of data that wastes capacity [7, 15, 75], and a lack of information in the second layer for decision making [7]. "Exclusiveness" is one of the main challenges. Either API changes for cooperation are required [24, 75] or some sort of hints from the upper layer needs to be propagated or derived [7, 45, 79, 80]. For instance, with DEMOTE [75], the lower level deletes a block from its cache when it is read by the upper level. Achieving exclusiveness in the application-kernel cache structure with one compressed layer would be an interesting future work.

Evolving storage devices (e.g., NVM) [16, 33, 41, 42, 44, 76] and use cases (e.g., S3) [25, 35, 62] have led to new techniques to manage storage hierarchies and cache cooperation. For example, EDT [25] decides and adapts data placement between tiers of SSDs and HDDs according to workload, aiming to minimize power consumption. D3N [35] also adapts sizes for multi-level caching with a ghost cache, but aiming to alleviate network imbalance. A whole-stack programmable caching scheme is proposed [62] with APIs for size allocation of caches in layers within multi-tenant data center. The adaptation space of Symbiosis, which accounts for computation (compression), capacity, and IO, is enlarged by modern fast block devices.

Our approach only tunes the sizes of caches and is optimized for the application-kernel cache structure, without altering their interaction. Notably, it does not require modifications to the OS kernel. These advanced communication techniques and policies are complementary.

Kernel Cache and Application Coordination: Deep understanding of kernel caching is crucial to performance optimization across the storage stack. The performance impact of kernel cache replacement policies and directory cache have been studied [10, 34, 66]. Butt *et al.* [11] build a simulator studying kernel prefetching. Tricache [21] replaces the kernel page cache for performance and also emphasizes transparent cache management for applications. Lee *et al.* [40] enable application-specific kernel caching. Our work, instead, utilizes simulation integrated into applications in a live system to adapt cache configuration.

7 Conclusion

We have introduced Symbiosis, a framework to enable robust cache adaptation for key-value storage systems. With careful study of the performance space, we develop an online simulator which enables a live key-value storage system to adapt its application cache size and achieve high performance. Across a wide range of workloads and settings, we demonstrate the overall benefits of our approach, as shown through implementations in three production key-value storage systems: LevelDB, WiredTiger, and RocksDB. We open source our framework, workloads traces, modified systems, and utilities to facilitate further investigation [1].

8 Acknowledgement

We thank Sam H. Noh (shepherd), anonymous reviewers, and the ADSL group for their comments and suggestions. This material was supported by funding from NSF CNS-1838733. Jing Liu was supported by a Meta PhD Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or other institutions.

References

[1] Symbiosis Repository. https://github.com/daiyifandanny/Symbiosis, 2023.

[2] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman. Principles of Optimal Page Replacement. *J. ACM*, 18(1):80–93, January 1971.

[3] Apache. Cassandra. http://cassandra.apache.org/.

[4] Apache. Kakfa. http://kafka.apache.org/.

[5] Ismail Ari, Ahmed Amer, Robert Gramacy, Ethan L. Miller, Scott A. Brandt, and Darrell D. E. Long. ACME: Adaptive Caching Using Multiple Experts. In *In Proceedings in Informatics*, pages 143–158, 2002.

[6] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*, New York, NY, June 2013.

[7] Lakshmi N. Bairavasundaram, M. Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, Munich, Germany, June 2004.

[8] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proceedings of the USENIX Annual Technical Conference (USENIX '19)*, Renton, WA, July 2019.

[9] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI '18)*, Renton, WA, April 2018.

[10] Nathan C. Burnett, John Bent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Management. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, Monterey, CA, June 2002.

[11] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms. In *Proceedings of the 2005 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '05)*, Banff, Canada, June 2005.

[12] Zhao Cao, Shimin Chen, Feifei Li, Min Wang, and X Sean Wang. LogKV: Exploiting Key-Value Stores for Event Log Processing. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[13] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, Virtual Conference, February 2020.

[14] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. Cosine: a Cloud-cost Optimized Self-designing Key-value Storage Engine. *Proceedings of the VLDB Endowment*, 15(1):112–126, 2021.

[15] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction-based Cache Placement for Storage Caches. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, San Antonio, Texas, June 2003.

[16] Wonil Choi, Bhuvan Urgaonkar, Mahmut Kandemir, Myoungsoo Jung, and David Evans. Fair Write Attribution and Allocation for Consolidated Flash Cache. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, Lausanne, Switzerland, March 2020.

[17] 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 '10)*, Indianapolis, IA, June 2010.

[18] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*, Virtual Conference, February 2021.

[19] Maria R. Ebling, Lily B. Mummert, and David C. Steere. Overcoming the Network Bottleneck in Mobile Computing. In *1994 First Workshop on Mobile Computing Systems and Applications*, pages 34–36, 1994.

[20] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. In *Proceedings of the 24th International Symposium on High Performance Computer Architecture (HPCA-18)*, Vienna, Austria, February 2018.

[21] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. TriCache: A User-Transparent Block Cache Enabling High-Performance Out-of-Core Processing with In-Memory Programs. In *Proceedings of the 16th USENIX Conference on Operating Systems Design and Implementation (OSDI '22)*, Carlsbad, CA, July 2022.

[22] Michael R. Frasca and Ramya Prabhakar. SRC: Virtual i/o Caching: Dynamic Storage Cache Management for Concurrent Workloads. In *International Conference on Supercomputing (ICS '11)*, Tucson, Arizona, May 2011.

[23] Sanjay Ghemawhat, Jeff Dean, Chris Mumford, David Grogan, and Victor Costan. LevelDB. https://github.com/google/leveldb, 2011.

[24] Binny S. Gill. On Multi-level Exclusive Caching: Offline Optimality and Why Promotions Are Better Than Demotions. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, San Jose, CA, February 2008.

[25] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost Effective Storage using Extent Based Dynamic Tiering. In *Proceedings of the 9th USENIX Symposium on File and Storage Technologies (FAST '11)*, San Jose, CA, February 2011.

[26] Tejun Heo, Dan Schatzberg, Andrew Newell, Song Liu, Saravanan Dhakshinamurthy, Iyswarya Narayanan, Josef Bacik, Chris Mason, Chunqiang Tang, and Dimitrios Skarlatos. IOCost: Block IO Control for Containers in Datacenters. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, Lausanne, Switzerland, February 2022.

[27] David A. Holland, Elaine Angelino, Gideon Wald, and Margo I. Seltzer. Flash Caching on the Storage Client. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, CA, February 2013.

[28] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache. In *Proceedings of the USENIX Annual Technical Conference (USENIX '15)*, Santa Clara, CA, July 2015.

[29] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. Improving Flash-Based Disk Cache with Lazy Adaptive Replacement. *ACM Trans. Storage*, 12(2), 2016.

[30] Stratos Idreos and Mark Callaghan. Key-Value Storage Engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*, Portland, OR, June 2020.

[31] ArangoDB Inc. What's the Latest with ArangoDB? `https://github.com/arangodb/arangodb`, 2022.

[32] PingCap. Inc. TiDB Introduction. `https://docs.pingcap.com/tidb/dev/overview`, 2022.

[33] Dejun Jiang, Yukun Che, Jin Xiong, and Xiaosong Ma. uCache: A Utility-Aware Multilevel SSD Cache Management Policy. In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 391–398, 2013.

[34] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Localities. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, San Francisco, CA, December 2005.

[35] Emine Ugur Kaynar, Mania Abdi, Mohammad Hossein Hajkazemi, Ata Turk, Raja R. Sambasivan, David Cohen, Larry Rudolph, Peter Desnoyers, and Orran Krieger. D3N: A multi-layer cache for the rest of us. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 327–338, 2019.

[36] Ricardo Koller, Ali José Mashtizadeh, and Raju Rangaswami. Centaur: Host-Side SSD Caching for Storage Performance Control. In *2015 IEEE International Conference on Autonomic Computing (ICAC '15)*, Grenoble, France, July 2015.

[37] Ricardo Koller, Akshat Verma, and Raju Rangaswami. Estimating Application Cache Requirement for Provisioning Caches in Virtualized Systems. In *Proceedings of the 19th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Washington, DC, July 2011.

[38] Kubernetes.io. What is cgroup v2. `https://kubernetes.io/docs/concepts/architecture/cgroups/#cgroup-v2`.

[39] Jaewon Kwak, Eunji Hwang, Tae-Kyung Yoo, Beomseok Nam, and Young-Ri Choi. In-Memory Caching Orchestration for Hadoop. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 94–97, 2016.

[40] Dusol Lee, Inhyuk Choi, Chanyoung Lee, Sungjin Lee, and Jihong Kim. P2Cache: An Application-Directed Page Cache for Improving Performance of Data-Intensive Applications. In *15th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '20)*, Boston, MA, July 2023.

[41] Eunji Lee and Hyokyung Bahn. Caching Strategies for High-Performance Storage Media. *ACM Trans. Storage*, 10(3), 2014.

[42] Eunji Lee, Hyojung Kang, Hyokyung Bahn, and Kang G. Shin. Eliminating Periodic Flush Overhead of File I/O with Non-Volatile Buffer Cache. *IEEE Transactions on Computers*, 65(4):1145–1157, 2016.

[43] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A Capacity-Optimized SSD Cache for Primary Storage. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, Philadelphia, PA, June 2014.

[44] Chu Li, Dan Feng, Yu Hua, and Fang Wang. Improving RAID Performance Using an Endurable SSD Cache. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 396–405, 2016.

[45] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. Second-Tier Cache Management Using Write Hints. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, San Francisco, CA, December 2005.

[46] Zhang Liu, Hee Won Lee, Yu Xiang, Dirk Grunwald, and Sangtae Ha. eMRC: Efficient Miss Ratio Approximation for Multi-Tier Caching. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*, Virtual Conference, February 2021.

[47] Thanos Makatos, Yannis Klonatos, Manolis Marazakis, Michail D. Flouris, and Angelos Bilas. Using Transparent Compression to Improve SSD-Based I/O Caches. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*, Paris, France, April 2010.

[48] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9, 1970.

[49] Fei Meng, Li Zhou, Xiaosong Ma, Sandeep Uttamchandani, and Deng Liu. vCacheShare: Automated Server Flash Cache Space Management in a Virtualization Environment. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, Philadelphia, PA, June 2014.

[50] Meta. RocksDB. http://rocksdb.org/.

[51] MongoDB. MongoDB WiredTiger. https://docs.mongodb.org/manual/core/wiredtiger/.

[52] Arjun Narayan and Peter Mattis. Why we built CockroachDB on top of RocksDB. https://www.cockroachlabs.com/blog/cockroachdb-on-rocksd/, 2019.

[53] Yuanjiang Ni, Ji Jiang, Dejun Jiang, Xiaosong Ma, Jin Xiong, and Yuangang Wang. S-RAC: SSD Friendly Caching for Data Center Workloads. In *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR '16)*, Haifa, Israel, June 2016.

[54] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Caching Less for Better Performance: Balancing Cache Size and Update Cost of Flash Memory Cache in Hybrid Storage Systems. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012.

[55] Zhu Pang, Qingda Lu, Shuo Chen, Rui Wang, Yikang Xu, and Jiesheng Wu. ArkDB: A key-value engine for scalable cloud storage services. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data (SIGMOD '21)*, Virtual Conference, June 2021.

[56] Andy Pavlo. It is too expensive/time-consuming to build a DBMS from scratch. https://twitter.com/andy_pavlo/status/1523666179247595520, 2022.

[57] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), 2014.

[58] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning Cache Replacement with CACHEUS. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*, Virtual Conference, February 2021.

[59] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. Dynamic Performance Profiling of Cloud Caches. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*, Seattle, WA, November 2014.

[60] Peter Shah and Keith Smith. Method for using service level objectives to dynamically allocate cache resources among competing workloads. https://patents.google.com/patent/US9836407B2/en, 2017.

[61] SQLite. SQLite transactional SQL database engine. http://www.sqlite.org/.

[62] Ioan Stefanovici, Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey. Software-Defined Caching: Managing Caches in Multi-Tenant Data Centers. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '15)*, Kohala Coast, HI, August 2015.

[63] Michael Stonebraker. Operating System Support for Database Management. *Commun. ACM*, 24(7), 1981.

[64] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *J. Supercomput.*, 28(1), 2004.

[65] Dan Tang, Yungang Bao, Weiwu Hu, and Mingyu Chen. DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA-10)*, Bangalore, India, January 2010.

[66] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E. Porter. How to Get More Value from Your File System Directory Cache. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, October 2015.

[67] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache Modeling and Optimization using Miniature Simulations. In *Proceedings of the USENIX Annual Technical Conference (USENIX '17)*, Santa Clara, CA, July 2017.

[68] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC Construction with SHARDS. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST '15)*, Santa Clara, CA, February 2015.

[69] Yang Wang, Jiwu Shu, Guangyan Zhang, Wei Xue, and Weimin Zheng. SOPA: Selecting the Optimal Caching Policy Adaptively. *ACM Trans. Storage*, 6(2), 2010.

[70] Abdul Wasay, Brian Hentschel, Yuze Liao, Sanyuan Chen, and Stratos Idreos. Mothernets: Rapid deep ensemble learning. *Proceedings of Machine Learning and Systems*, 2:199–215, 2020.

[71] Johannes Weiner. PSI - Pressure Stall Information. https://www.kernel.org/doc/html/latest/accounting/psi.html, April 2018.

[72] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, Lausanne, Switzerland, February 2022.

[73] Wikipedia. Data compression. https://en.wikipedia.org/wiki/Data_compression.

[74] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield. Characterizing Storage Workloads with Counter Stacks. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[75] Theodore Wong and John Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, Monterey, CA, June 2002.

[76] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*, Virtual Conference, February 2021.

[77] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. ZExpander: A Key-Value Cache with Both High Performance and Fewer Misses. In *Proceedings of the EuroSys Conference (EuroSys '15)*, Bordeaux, France, April 2015.

[78] Tim Xu. Quality of Service for Memory Resources. https://kubernetes.io/blog/2021/11/26/qos-memory-resources/, 2021.

[79] Gala Yadgar, Michael Factor, Kai Li, and Assaf Schuster. Management of Multilevel, Multiclient Cache Hierarchies with Application Hints. *ACM Trans. Comput. Syst.*, 29(2), 2011.

[80] Gala Yadgar, Michael Factor, and Assaf Schuster. Karma: Know-it-All Replacement for a Multilevel Cache. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, CA, February 2007.

[81] Feng Zhang, Weitao Wan, Chenyang Zhang, Jidong Zhai, Yunpeng Chai, Haixiang Li, and Xiaoyong Du. CompressDB: Enabling Efficient Compressed Data Direct Processing for Various Databases. In *Proceedings*

of the 2022 ACM SIGMOD International Conference on Management of Data (SIGMOD '22), Philadelphia, PA, USA, June 2022.

[82] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards Practical Page Coloring-Based Multicore Cache Management. In *Proceedings of the EuroSys Conference (EuroSys '09)*, Nuremburg, Germany, April 2009.

[83] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Transactions on parallel and distributed systems*, 15(6):505–519, 2004.

A Artifact Appendix

Abstract

Symbiosis is a framework for key-value storage systems that dynamically configures application and kernel cache sizes to improve performance. This artifact includes code for Symbiosis-integrated LevelDB, RocksDB, and WiredTiger, the offline simulator, scripts to run these applications, and several workload traces used in the paper.

Scope

Our artifact is fully functional, including all the features and optimizations mentioned in the design and three implementations (i.e., Section 4). We provide several traces used in our evaluation. Running the three storage engines with and without Symbiosis in similar hardware settings can support our findings of the cache-sizing problem and the effectiveness of Symbiosis's design and techniques.

The offline simulator can run various size configurations and workloads; its kernel cache can be configured to mimic the kernel cache behaviors (e.g., 2Q and read-ahead). Running the simulator experiments with the same configuration as Section 3 is expected to exactly reproduce the results, supporting our findings about the impacting factors and performance gain under various workloads.

Contents

We describe the contents of the subdirectories in the root of the repository as below:

- `leveldb` contains Symbiosis-embedded LevelDB that is used to reproduce experiments in Section 5.1.1, Section 5.2.2, and Section 5.3.
- `wiredtiger` contains Symbiosis-embedded WiredTiger that is used to reproduce experiments in Section 5.1.3.
- `rocksdb` contains Symbiosis-embedded RocksDB that is used to reproduce experiments in Section 5.1.4.
- `simulator` contains the cache simulator (in Python) used in Section 3.
- `traces` includes all the traces for the experiments mentioned above.
- `scripts` includes the scripts to run the experiments mentioned above. Detailed instructions can be found in `ae_readme.txt`.

Hosting

The artifact is hosted on `https://github.com/daiyifandanny/Symbiosis`, on branch *main* with commit id *36e3ea7*.

Requirements

Offline Simulations (Section 3):

- Software: Python 3.8. Python package numpy and simpy.

Performance Evaluation (Section 5):

- Library: sdt, zstd, and snappy. Installation guide can be found in `ae_readme.txt`.
- System: Linux kernel 5.11 and Ubuntu 20.04.
- Hardware: Hardware listed in Table 1, especially an OptaneSSD, is necessary for reproducing the exact results. With different hardware, offline calibration of the application and kernel cache miss costs is required; the result (in microsecond) needs to be set in `leveldb/util/adapter.h`.

Optimizing File Systems on Heterogeneous Memory by Integrating DRAM Cache with Virtual Memory Management

Yubo Liu[1], Yuxin Ren[1], Mingrui Liu[1], Hongbo Li[1], Hanjun Guo[1], Xie Miao[1],
Xinwei Hu[1], and Haibo Chen[1,2]
[1]*Huawei Technologies Co., Ltd.* [2]*Shanghai Jiao Tong University*

Abstract

This paper revisits the usage of DRAM cache in DRAM-PM heterogeneous memory file systems. With a comprehensive analysis of existing file systems with cache-based and DAX-based designs, we show that both suffer from suboptimal performance due to excessive data movement. To this end, this paper presents a cache management layer atop heterogeneous memory, namely FLAC, which integrates DRAM cache with virtual memory management. FLAC is further incorporated with two techniques called zero-copy caching and parallel-optimized cache management, which facilitates fast data transfer between file systems and applications as well as efficient data synchronization/migration between DRAM and PM. We further design and implement a library file system upon FLAC, called FlacFS. Micro benchmarks show that FlacFS provides up to two orders of magnitude performance improvement over existing file systems in file read/write. With real-world applications, FlacFS achieves up to 10.6 and 9.9 times performance speedup over state-of-the-art DAX-based and cache-based file systems, respectively.

1 Introduction

Emerging persistent memory (*e.g.*, 3DXPoint [14, 26] and CXL-based SSD [18]) promise fast and byte-addressable accesses to large volume of data. This brings a trend of deploying heterogeneous memory of a volatile memory layer (DRAM) and a persistent memory layer (PM). However, it raises a natural question: how to maximize performance atop such a heterogeneous architecture?

State-of-the-art file systems for heterogeneous memory can mainly fall into two categories: using DRAM as a cache for PM (DRAM cache) or providing direct access (DAX) to PM. Caching pages in DRAM, such as the VFS page cache, is a common design in traditional file systems (*e.g.*, EXT4 and XFS [44]) to bridge the performance gap between fast DRAM and slow persistent storage devices (*e.g.*, HDD and SSD). However, many previous studies [10] show that DRAM cache incurs significant overhead under the fast, all-memory architecture. Therefore, most existing systems (*e.g.*, NOVA [51], SplitFS [20], and ctFS [31]) resort to DAX , which bypasses the DRAM cache and performs I/Os on PM directly.

However, DAX is still suboptimal for heterogeneous memory file systems. First, the performance gap between PM and DRAM cannot be ignored in the present and future (the PM latency may range from hundreds to thousands of nanoseconds [18], which is much higher than DRAM). Such high PM latency easily limits the file system performance. Second, DAX potentially loses the performance benefit of data locality provided by the DRAM cache. According to our analysis, the performance of DAX-based systems is inferior to that of DRAM cache systems in scenarios with high concurrency and strong data locality, even though the VFS page cache framework introduces high software overhead. Last but not least, instant persistence is the best scene of DAX; but it is an overkill in many real-world scenarios [49].

To this end, this paper revisits the usage of DRAM cache in heterogeneous memory architecture. According to our quantitative analysis, we summarize two challenges of building an efficient cache framework on heterogeneous memory:

Challenge 1. Data transfer overhead between application buffer and DRAM cache is high. Transferring data between the application and DRAM cache is the most critical fast-path operation; but existing cache frameworks use memory copy that introduces substantial performance overhead. Our experiments show that data copying occupies up to 84% of the overhead in the file system with the VFS page cache.

Challenge 2. The impact of "cache tax" is significant. In addition to data transfer, existing cache frameworks spend lots of effort to synchronize (flushing dirty data) and migrate data across DRAM cache and PM (moving data into/out of cache). Currently, such operations are implemented in a synchronous and sequential way and significantly increase performance penalty (more than 30%).

We argue that the main reason is that existing cache frameworks (*e.g.*, VFS page cache) are built upon the virtual memory subsystem, which makes it difficult to avoid the cache-application data copying and hide the overhead of cross-layer data synchronization/migration. Hence, this paper advocates an integration of DRAM page cache into virtual memory management of operating systems and proposes FLAC (**FLA**t **C**ache), a novel cache framework for heterogeneous memory. FLAC provides a single-level address space of heterogeneous memory. File system developers can leverage the exposed interfaces to the data store on FLAC to enjoy the efficient DRAM cache on data I/O paths (other modules of file system are independent of FLAC). FLAC further builds two novel techniques to deal with the two challenges outlined above:

1) Zero-Copy Caching. FLAC proposes the heterogeneous page table that unifies heterogeneous memory into a single level. Virtual pages within FLAC can be dynamically mapped to physical pages on DRAM or PM according to their states (*i.e.*, cached or evicted). We then design the page attaching mechanism, a set of tightly coupled management operations on the heterogeneous page table, which optimize the data transfer between applications and cache in a zero-copy manner. The core idea of page attaching is to map pages between source and destination addresses with enforced copy on write (COW). As a result, data read/write to/from FLAC is executed by page attaching to realize efficient and safe data transfer.

While page remapping optimizations are also used in some systems to reduce the overhead of data copy [9, 30, 38, 40], simply adopting this idea in the file system cache faces some unique challenges. First, FLAC addresses the side-effects of page unaligned and expensive COW page fault by the sliding window buffer and batch faulting/detaching, respectively. Second, the zero-copy caching makes the page have multiple versions, and it requires FLAC to have a new cache management mechanism to ensure data consistency and high concurrency.

2) Parallel-Optimized Cache Management. The cache management mechanism of FLAC must ensure a low "cache tax" impact. Leveraging the multi-version feature brought by the zero-copy caching, FLAC fully exploits the parallelism of data synchronization and migration with critical I/O paths. FLAC proposes the 2-Phase flushing that allows the expensive persistence phase in dirty data synchronization to be lock-free, and proposes the asynchronous cache miss handling to amortize the overhead of loading data to cache in the background.

To demonstrate the effectiveness of FLAC, we design and implement FlacFS, a file system for building its data store on FLAC. Evaluation shows that FlacFS provides a performance increase of more than two orders of magnitude over state-of-the-art DAX-based and cache-based file systems in the micro benchmarks. With real-world applications, FlacFS achieves up to 10.6 and 9.9 times performance speedup over DAX-based and cache-based systems, respectively.

The contributions of this paper include:

- It quantitatively analyses the cache and DAX frameworks and summarizes the key challenges of cache framework design on heterogeneous memory.
- It designs and implements FLAC, a novel cache framework for heterogeneous memory file systems that including the techniques of zero-copy caching and parallel-optimized cache management.
- It implements a file system (FlacFS) based on FLAC, and demonstrate the benefits via micro/macro benchmarks and real-world applications.

The rest of this paper is organized as follows: Section 2 introduces the background and motivation; Section 3 presents the key designs of FLAC; Section 4 introduces the implementation of FlacFS; Section 5 discusses the limitations and challenges; Section 6 shows the detailed evaluation of FLAC/FlacFS; Section 7 concludes the paper.

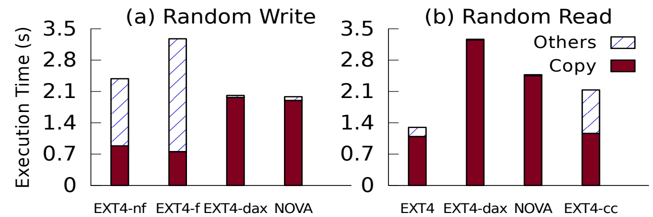


Figure 1: Traditional Cache *vs.* DAX. "-f/-nf": with/without background flushing; "-cc": cold cache.

2 Background and Motivation

2.1 Heterogeneous Memory

With the emergence of new persistent storage media (*e.g.*, 3DXPoint [14], CXL-based SSD [18, 23, 55, 56]), the storage architecture evolves from memory-block to all-memory. A typical heterogeneous memory architecture consists of a fast, volatile, small capacity layer (DRAM), and a slow, non-volatile, large capacity layer (PM). Different types of memories present heterogeneity in multiple aspects [33]. *1) Latency Gap.* The latency of DRAM is about tens of nanoseconds, while the latency of low-level memory range from hundreds to thousands of nanoseconds [18, 34]. *2) Bandwidth Gap.* The bandwidth of DRAM can reach tens of GB, while it is only about a few GB of existing PM [52]. *3) Concurrency Gap.* The PM has lower concurrency than the DRAM [11, 20]. For example, existing PM hardware based on 3DXPoint is hard to scale beyond 4 concurrent [16] in the single channel. In summary, the performance gap between DRAM and PM and between different types of PMs cannot be ignored, which make it challenging to design efficient storage systems for heterogeneous memory.

2.2 Direct Access (DAX) *vs.* Cache

Heterogeneous memory raises an important question for file system designers: what kind of storage framework can take advantage of different memory devices? There are two typical storage frameworks are used on heterogeneous memory: 1) traditional page cache based on the DRAM-block device architecture (*i.e.*, VFS page cache) and 2) direct access (DAX). We quantitatively analyze three typical file systems with the VFS page cache (EXT4) and DAX (EXT4-DAX [7], NOVA [51]) by performing random writes/reads on a 10GB file with 2MB I/O (the testbed is introduced in §6). Three important observations are found from our experiments:

Observation 1*: Existing DAX and cache frameworks are sub-optimal, and DRAM cache still has great value for heterogeneous memory file systems.*

The VFS page cache is a typical cache framework that is designed to bridge the performance gap between DRAM and

block devices. However, the VFS page cache has a heavy software stack, which makes it unsuitable for the heterogeneous memory structure. Therefore, many heterogeneous memory file systems proposed in the past decade resort to the DAX method, *i.e.*, bypassing the DRAM cache in the data I/O path. However, we think DRAM cache still has a lot of value in heterogeneous memory file systems. First, the performance gap between PM and DRAM cannot be ignored. Figure 1 shows that the VFS page cache still has better performance than DAX in some cases (*e.g.*, read). Second, taking advantage of data locality is an effective method of performance optimization, but DAX misses this opportunity. Third, POSIX is still a mainstream semantics and it can tolerate cached I/Os, which makes instant persistence in DAX an overkill in many real-world scenarios [49].

Observation 2*: Data transfer overhead between the file system and the application buffer is significant but often overlooked, and it is one of the keys to unlocking the potential of the cache in heterogeneous memory.*

Data I/Os (file read/write) need to transfer data between the application buffer and the storage system (cache space or persistent data space). Memory copy is the mainstream method to transfer data, but in our experiment, it takes up more than 23% and 96% of the total overhead in cache-based and DAX-based file systems, respectively. In particular, the performance bottleneck of data copy between cache and application is obvious in heterogeneous memory systems since the latency of PM is much lower than traditional block devices.

Observation 3*: The "Cache Tax" in traditional cache frameworks is heavy, and it mainly includes the overhead of data synchronization and migration.*

Caching increases storage levels and brings extra data management overhead. Figure 1 shows that the "cache tax" (denoted as other) takes up to 77% of the execution time in EXT4. Figure 2 shows the core processes of the typical DRAM page cache, which reveals the composition of the "cache tax". From our experiments, the data synchronization (background dirty flushing) and data migration (cache miss handling) lead to 37% and 65% performance declines, respectively.

2.3 Motivation

According to the previous analysis, an efficient heterogeneous memory cache framework needs to meet two requirements: 1) low application-cache transfer overhead and 2) low "cache tax" impact. However, exiting cache frameworks (*e.g.*, VFS page cache) do not fully exploit the potential of DRAM cache in heterogeneous memory systems. They are difficult to avoid the data copy between the DRAM cache and the application buffer. At the same time, they are difficult to transparently overlap the critical I/O paths and the cross-layer data synchronization/migration. The motivation of this work is to integrate the DRAM page cache with the virtual memory management subsystem, and it brings two key principles for our design.

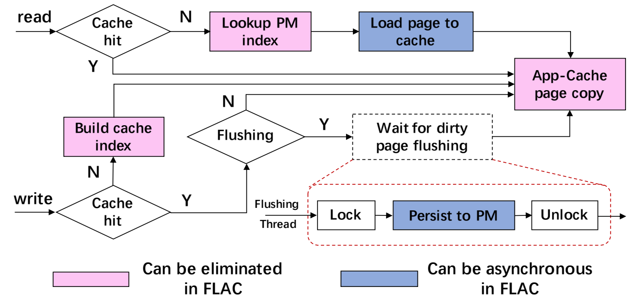


Figure 2: Typical Diagram of Page Cache.

Principle 1: *Optimizing data transfer between the cache and the application by zero-copy.* Traditional DRAM cache frameworks simply take advantage of the performance advantages of DRAM, but ignore another important advantage of DRAM cache: it is homogeneous with the application runtime. By co-designing the DRAM cache and the virtual memory subsystem, the data copy during application-cache transfer can be avoided by page mapping. FLAC proposes the zero-copy caching technique to avoid application-cache data copy and redundant indexes (red squares in Figure 2).

Principle 2: *Reducing the impact of "cache tax" by hiding the data synchronization/migration overhead.* The "cache tax" is difficult to eliminate, but their impact on the critical I/O paths can be reduced by improving the parallelism between the data synchronization/migration and the front-end I/Os. FLAC proposes the parallel-optimized cache management mechanism to amortize the data synchronization/migration overhead in the background (blue squares in Figure 2).

3 FLAC Design

3.1 Overview

This work proposes FLAC, a **FLA**t **C**ache framework integrated with the virtual memory subsystem to deeply explore the potential of cache for heterogeneous memory systems. As shown in Figure 3, FLAC maintains a range of contiguous virtual memory addresses, called FLAC space. The size of FLAC space is equal to the usable PM space, and it provides the data storage area with the built-in DRAM page cache for the heterogeneous memory file system. The FLAC space is indexed by the heterogeneous page table, which makes page physical locations transparent and exposes a single-level memory space to file system developers. Data is transferred between the application and the FLAC space with the zero-copy approach (§3.2), and synchronized/migrated between DRAM and PM with the parallel-optimized mechanism (§3.3). Table 1 shows the main APIs of FLAC.

`init_flac`: This API is used to initialize and bind the given PM to the FLAC space for file data storage. If the FLAC space has already been created on the PM, it rebuilds the FLAC space from the last consistent state.

`zcopy_from/to_flac`: The file system based on FLAC internally uses these two APIs to transfer data and support

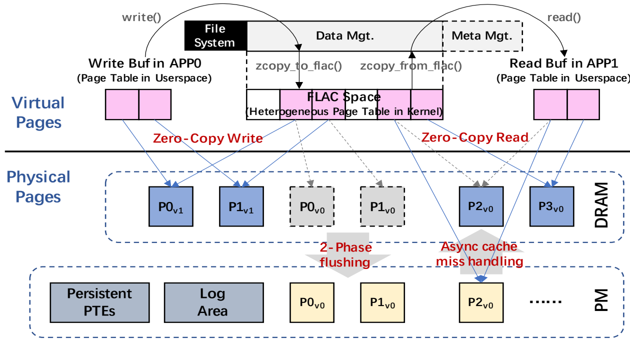


Figure 3: Architecture of FLAC. File system runs the data management on top of FLAC. Application accesses data by file read/write, and they are converted to the zero-copy transfer APIs in FLAC. Data is stored in a flat memory address, which is transparent to the physical locations of the pages through heterogeneous page table.

file read/write operations, which are similar to the action of `copy_to/from_user` in traditional kernel file systems.

`pflush_add/commit`: This pair of APIs are used to explicitly flush dirty data from DRAM to PM, and they give developers the flexibility to customize flushing policies. Dirty pages are added to a flush handle (`pflush_add`) and flushed to PM in a transaction (`pflush_commit`). File systems use the *fs_metalog* parameter to ensure the consistency of FS-level metadata during data flushing.

`pfree`: This API is used to atomically reclaim a range of FLAC space. It invalidates the page on the DRAM/PM and removes the page table mapping.

Architecture and Usage. FLAC runs under the file system as a development framework. Developers of heterogeneous memory file systems customize the file data management on the FLAC space (*e.g.*, file read/write logic and data flushing policy) by encapsulating the APIs above, and applications access the data on the FLAC space by normal file interfaces. The other modules of the file system (*e.g.*, metadata management) are independent to FLAC, which can be flexibly designed and implemented. FLAC's APIs can be called by `ioctl`s or kernel functions, which allows developers to flexibly implement file systems in the userspace or kernel.

3.2 Zero-Copy Caching

3.2.1 Heterogeneous Page Table

As Figure 3 shows, FLAC uses the heterogeneous page table, a customized sub-level table (including one or multiple PUDs) of the kernel page table, to maintain the FLAC space: it is a range of consecutive kernel virtual memory addresses and its size is equal to the usable PM size. The positions of pages (DRAM/PM) in the FLAC space are transparent for the file systems running upon it. This design has two meanings: **1)** The address indexed in the page table is dynamically mapped to DRAM or PM as the page is cached or evicted, and a bit in the PTE is used to indicate the location of the page. **2)** Page table entries (PTEs) belonging to the FLAC space are

Table 1: Main APIs of FLAC (for file system developer)

API	Main Para.	Description
`init_flac`	pm_path	Create/Recover the FLAC space
`zcopy_from_flac` `zcopy_to_flac`	from_addr to_addr size	Zero-copy transfer data between the application and the FLAC space
`pflush_add`	pflush_handle addr size	Attach (map) the pages to the flushing buffer and add to the handle
`pflush_commit`	pflush_handle fs_metalog	Flush the pages in the handle and update the metadata atomically
`pfree`	addr size fs_metalog	Reclaim the PM pages and update the metadata atomically

replicated in PM for fault recovery. The heterogeneous page table unifies the page indexes of cache and persistent storage and simplifies cache access and management.

PM is divided into three areas. **1)** The persistent PTE area records the mapping information between the virtual addresses and the PM pages. All PTEs of the heterogeneous page table are mirrored on PM. When a page is flushed from DRAM to PM, FLAC records the related offset in the PM device to the persistent PTE for recovery. **2)** The log area logs the modifications of FLAC-level (*e.g.*, persistent PTE) and FS-level metadata (*e.g.*, inode) by the FS-FLAC collaboration logging mechanism when the persistent data modification APIs (`pflush_commit/pfree`) are called. **3)** The page area contains multiple 4KB units for file data storage. Data pages are persisted in this area during flushing.

Developers call `init_flac` to prepare the FLAC space and it is responsible for rebuilding the heterogeneous page table. First, FLAC checks the logs to determine whether the system exits abnormally, and if so, recovers it to the last consistent state. Then, the PGDs, PUDs, PMDs, and PTEs of the heterogeneous page table are created in DRAM. In particular, the locations in PTEs are rebuilt by translating the related offsets in the persistent PTEs (if have) to the physical location of the PM pages. After initialization, all valid pages on PM are mapped to the heterogeneous page table.

3.2.2 Transfer Data with Page Attaching

The core technique used to achieve the goal of zero-copy is a new virtual memory management operation – page attaching (Interface (1)). The `attach` includes four parameters: two address and their size, and the permission mode. Page attaching maps the pages of the source address (*from_addr*) to the destination address (*to_addr*) with the given size. The permission mode (*pmode*) allows users to set permissions on source and destination addresses after page attaching (*e.g.*, read-only).

Page attaching first searches the PTEs of source and destination addresses then maps the physical pages of the source

address to the destination address (the permission and page reference counter are also set) and finally flushes the TLB. It will be aborted if the source addresses are not faulted (*i.e.*, mapped to the physical pages), but there is no restriction on the destination addresses. If the destination addresses are faulted (*e.g.*, overwrite), the reference counter of the old physical pages will be reduced and they are reclaimed by the memory subsystem when their counters reach 0.

$$attach(to_addr, from_addr, size, pmode) \quad (1)$$

The APIs of `zcopy_from/to_flac` encapsulate `attach` for transferring data between the application and the FLAC space. For data security and isolation, the operated pages are set to read-only after attaching, so that subsequent writes on these pages will transparently trigger copy-on-write (COW) page fault, which ensures that the memory operations inside the application do not affect the data that have been mapped to the global cache and other applications. In particular, benefiting from the heterogeneous page table, pages can be attached whether they are cached or not and this feature delivers the design of asynchronous cache miss handling.

Handling Page Unaligned. The file I/O (<fd, offset, size>) is translated to the address and size of the FLAC space by the upper-layer file system, and the data is transmitted between the FLAC space and read/write buffer by page attaching (`zcopy_from/to_flac`). Page attaching requires that the operated addresses and sizes are page aligned, but file I/Os and buffers are arbitrary, resulting in the page unaligned problem. Our solution is to attach all the pages containing the required data and use a cursor to locate valid data in the application buffer. FLAC requires the upper-layer file system to ensure that the start address of each file is page aligned. Given an unaligned file I/O and a buffer, we first need to extend the file I/O to the FLAC space range that contains all the required pages, which is achieved by the automatic alignment mechanism. In addition, we need to ensure that the buffer is large enough, page aligned, and can represent the valid data in it, which is achieved through the sliding window buffer technology.

Automatic Alignment. The `zcopy_from/to_flac` check whether the given FLAC address and size are page aligned, and if it doesn't, the access range (start and end addresses) is automatically extended to page aligned. Then, the page attaching is executed. In particular, if the destination space is already mapped to the (old) pages, the hole(s) caused by automatic alignment is filled with the data in the old page(s) through copy after attaching.

Sliding Window Buffer. As Figure 4 shows, the application allocates (`swbuf_alloc`) the sliding window (SW) buffer by using the read/write size (*dsize*). It includes a `swbuf` structure and a page unaligned space (**bhead*) in the size of $\lceil dsize/4096 \rceil + 1$ pages (*bszie*). The application uses the **bhead* in SW buffer to serve file read/write with arbitrary offset and size. However, the valid data may not

Figure 4: Sliding Window (SW) Buffer. Application uses SW buffer to serve page unaligned read/write. The sliding window is used to identify valid data in the buffer.

start with the **bhead* due to the automatic alignment in `zcopy_from/to_flac`. Before using the data in the read-/write buffer, the application is asked to call the `swbuf_slide` to calculate the window of valid data in the SW buffer by using the file offset and **bhead*. The head of valid data is recorded in the **data*. It is worth noting that the SW buffer is not mandatory if the application can guarantee the offset and size of file read/write are page aligned.

Reducing COW Page Fault Overhead. The zero-copy data transfer ensures security and isolation by setting the source and destination memory read-only, and this makes the first write operation (`store` instruction) to the source (write buffer) or destination (read buffer) memory after `attach` to trigger COW page fault. According to our analysis, the main overhead of COW page fault includes two aspects: TLB flush and data copy. FLAC proposes two techniques to reduce the impact of COW page fault for different use cases.

Batch Fault. In some scenarios, applications directly process data in the read/write buffer, which causes a large number of pages in the buffer to be faulted with COW. FLAC optimizes this unfriendly case by executing the COW page faults in batch, thus reducing the number of TLB flushes. Batch faulting copies the data from the original pages to the new pages in batch and only needs to flush the TLB once. The application can call the `bfault` API for the read/write buffer before the data in the buffer are processed.

Detach. In some scenarios, the application just wants to reuse the read/write buffer's space instead of its data, which is a false sharing scenario (*e.g.*, pre-allocating a log buffer and reusing it after it is written to the storage system). To avoid COW page fault in this scenario, FLAC provides the `detach` API to remap the addresses of the read/write buffer to some new anonymous pages to absorb subsequent memory operations. The application can call the `detach` before the read/write buffer is reused. After detaching, the subsequent memory writes to the buffer will not trigger COW page faults.

3.3 Parallel-Optimized Cache Management

Due to the zero-copy caching design, FLAC requires a cache management mechanism for its multi-version feature, while ensuring a low "cache tax" impact. Fortunately, the multi-version feature and the heterogeneous page table design of

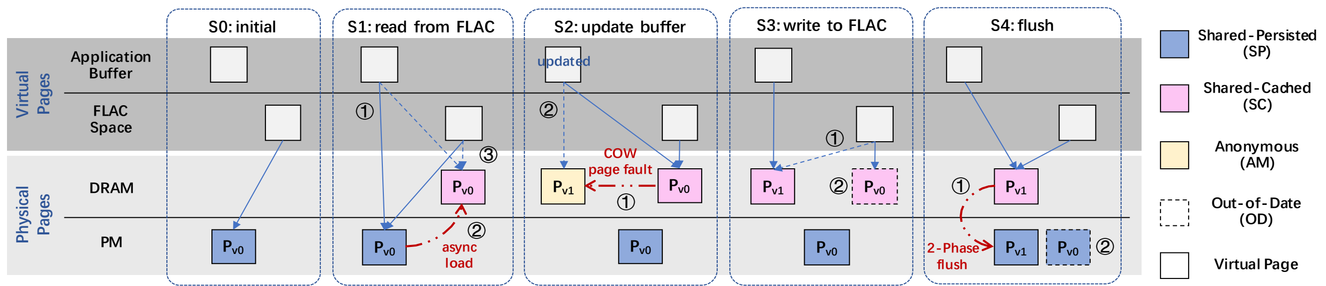


Figure 5: Page State/Version Transition. Solid blue arrow: current mapping; Blue dashed arrow: future mapping; Red dashed arrow: data copy.

FLAC allow us to fully exploit the parallelism of data synchronization/migration with critical I/O paths.

3.3.1 Parallel-Optimized Synchronization/Migration

Existing cache frameworks execute cache flushing and cache miss handling with large synchronization and migration overhead: Cache flushing locks the dirty pages until they are completely flushed, which blocks the front-end writes and dramatically reduces the performance; Cache miss handling blocks the I/Os until the pages are loaded to the DRAM cache. They are optimized by the following two techniques.

2-Phase Flushing. FLAC splits the dirty pages flushing into two phases: collection (`pflush_add`) and persistence (`pflush_commit`). The collection phase adds the given dirty pages to a flush handle, which allocates a fresh virtual memory address space as a temporary flush buffer and attaches the dirty pages to it. This phase requires a lock to prevent concurrent writes from modifying the target pages. The persistence phase is responsible for persisting the dirty pages in the flush handle to PM. This phase is lock-free since there are no concurrent accesses to the temporary buffer. Because the page mapping in the collection phase is much faster than cross-layer copy, the 2-Phase flushing mechanism significantly reduces the blocking time on concurrent writes due to dirty page synchronization (*e.g.*, background flushing).

The persistence phase is atomic. It flushes data pages by the log-structured method, *i.e.*, dirty data is written to the new PM pages and the out-of-date PM pages are reclaimed. The persistent PTEs are updated after dirty data is successfully flushed. The modifications of the PM page allocator and persistent PTEs are logged to ensure crash consistency. In addition, file systems may require FS-level metadata updates and data persistence to be the same transaction. FLAC provides the FS-FLAC collaboration logging mechanism to meet this goal (§3.3.4).

Asynchronous Cache Miss Handling. Cache miss has less impact on write operation because it does not require pages to be loaded into the cache (except in the case of page misalignment), but it is expensive on read operation. Benefiting from the heterogeneous page table, FLAC can directly attach the PM pages to the read buffer (returns immediately) and handle the cache miss asynchronously. A background thread in FLAC is responsible for loading the missed pages to DRAM and remapping the PTEs of FLAC space and application buffer(s) pointing to those PM pages to the cached DRAM pages. The page may have been modified to trigger the COW page fault before it is loaded to DRAM, which means it has the newest version in DRAM. The asynchronous cache miss handling checks if the page already has a new version in DRAM and skips if it does. This design makes it possible for the overhead of handling cache misses to be amortized in the background, thereby reducing the data I/O latency.

3.3.2 Page State/Version Transition

The page may have different states and versions in FLAC. There are four states of a page in FLAC: shared-persistent (SP), shared-cached (SC), anonymous (AM), and out-of-date (OD). SP and SC pages are stored in the global areas (PM data page area/DRAM cache) and are read-only; AM pages are readable and writable, which is the same as the normal anonymous page in processes; OD pages are invisible to the file system on FLAC and are managed by FLAC's reclamation mechanism. Figure 5 shows an example of page state/version transition through a sequence of operations. As the initial stage (S0), we assume that there is a page on the FLAC space and it is in the PM data page area.

Stage 1: Read from FLAC. The target page is an SP page, so cache miss happens when the application reads the page from the FLAC space to the application buffer (by the file system interface). FLAC first maps the buffer to the SP page and the read operation is returned (①). Then, the target page is asynchronously loaded to DRAM as an SC page (②) and the virtual pages of application buffer and FLAC space are remapped to the new SC page (③).

Stage 2: Update the buffer. When the application tries to update (by `store` instruction) the data in the buffer, a new version AM page is created by COW (①), and then it is mapped to the buffer address to absorb the updates (②). COW page fault is only triggered at the first time the SP/SC page (depending on if it is cached) is updated by the application, and subsequent memory accesses will directly perform on the AM page. In addition, the old version page in FLAC is still in its original state (SP/SC).

Stage 3: Write to FLAC. When the application writes (attaches) the page back to FLAC, the state of the page mapped

by the buffer is changed from AM to SC (①). The state of the old page that is mapped to the target address in FLAC will be changed to OD and reclaimed by FLAC when it is clean (②).

Stage 4: Background flush. The page will be synchronized to PM when the background flushing is triggered, which is implemented by the upper-layer file system. During background flushing, a new SP page is created (①) and the old version of the SP page is reclaimed by FLAC after the data is successfully synchronized (②).

Page State Semantics. Page state is at the process granularity as FLAC maintains the state by manipulating the process' page table. The FLAC-based file system has the same semantics in file read/write operations as traditional file systems. After one thread attaches (by file read/write) a page with SC/SP state, other threads in the same process can read it consistently. Once the attached page is modified and causes COW to generate an AM page, it can be shared within the process. FLAC currently does not support read/write shared pages between different processes (*e.g.*, using the FLAC space as inter-process shared memory). FLAC enforces mandatory isolation and COW in different processes will generate separate AM pages. However, FLAC may support this case by considering reverse page mapping during COW.

Durability. FLAC does not restrict the durability model, which is defined and implemented by the upper-layer file system. Our prototype FLAC-based system (FlacFS) uses the same durability model as traditional file systems. The cached data is persisted to PM under two cases, *i.e.*, the background flushing is triggered and the `fsync` is called by the file system user. FlacFS implements background flushing and `fsync` by encapsulating the data synchronization operations of FLAC (`pflush_add/commit`), and FLAC guarantees that these operations are atomic and recoverable by the FS-FLAC collaboration logging.

3.3.3 Cache Policy

The size of FLAC space is equal to the usable PM size, but the maximum DRAM cache usage is controllable and page eviction is triggered when the cache is full. Due to the zero-copy design naturally brings the advantage of deduplication, FLAC counts the pages that are *only* mapped by the FLAC space into the used size, and the pages that are mapped by both the FLAC space and application buffer are treated as an in-process pages (without taking up the cache space).

As this work mainly focuses on the cache framework, we just design a simple cache policy in our prototype, *i.e.*, it selects pages for eviction with the round-robin approach. Existing cache algorithms [17, 35, 41, 53, 54] can also be used for FLAC. For the sake of simplicity, a page can be evicted only when two conditions are met. First, the page is clean, *i.e.*, it has been synchronized to PM by background flushing or `fsync`. Second, the reference counter of the page is 1, which means that the page is only mapped by the FLAC space and not used by any application. After a page is evicted to PM, the target PTE of the FLAC space is remapped to the PM page and the DRAM page will be reclaimed.

In particular, the multi-version feature of FLAC does not incur additional space overhead compared to traditional page cache. The new version of the page being created in the application process by COW page fault, the new version does not take up space in the page cache before it is overwritten to FLAC. After overwriting, the virtual address of FLAC space is mapped to the new version and the old version is reclaimed.

3.3.4 FS-FLAC Collaboration Logging

For normal shutdown, the recovery process of FLAC only needs to rebuild the heterogeneous page table according to the persistent PTEs. For an unexpected shutdown, FLAC must recover the system to the last consistent state. As described in the 2-Phase flushing, persistent data modifications in FLAC (`pflush_commit/pfree`) are atomic. However, along with data modifications, the file system upon FLAC may need to update the related FS-level metadata (*e.g.*, page index) on PM in the FS-level atomic operation (*e.g.*, append).

To ensure complete consistency, FLAC provides the FS-FLAC collaboration logging mechanism to allow the data modifications in FLAC and FS-level metadata updates in a transaction. It requires the file system to make two efforts: **1)** File system should provide the self-formatted metadata log (*fs_metalog* parameter) when the persistent data modification APIs are called. FLAC concatenates the internal (FLAC-level) and external (FS-level) metadata log into an entry and appends it to the log area after a successful persistent data modification operation. **2)** File system should overload an external metadata recovery function provided by FLAC. During recovery, FLAC first commits the internal metadata log and then calls the external recovery function to commit the external metadata. After all logs are committed, FLAC can be recovered as normal shutdown.

4 Case Study: FlacFS

We implement FlacFS, a file system based on FLAC to show the usage and benefits of FLAC. FlacFS contains three additional designs, the metadata management, data management, and mechanism of security and consistency. FlacFS is a library file system implemented through memory semantics. Figure 6 shows the architecture of FlacFS. As FlacFS focuses on the cache framework, we draw from existing works on designs in some aspects.

4.1 Metadata Management

The metadata area is mapped as traditional shared memory on userspace. It includes two separate virtual memory addresses for DRAM and PM, while they are created by `shmget` and `mmap`, respectively. The metadata (inode) of the directory/file

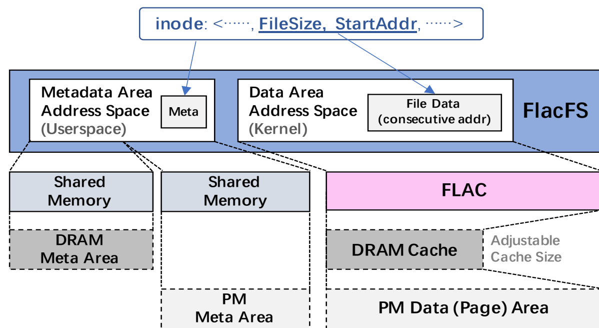


Figure 6: Implementation of FlacFS. The data space is built on FLAC, and file read/write are implemented by encapsulating FLAC's APIs. The metadata space is built on traditional shared memory. All inodes are cached in the DRAM hash table using their paths as keys, and their copies are persistently stored on PM.

is treated as a KV pair and stored in the inode hash table on shared memory using its full path as the key. The inode table is stored on both DRAM and PM to accelerate metadata operations. The metadata operations are performed in the DRAM inode table immediately and flushed to the PM inode table when the dirty data of the related files are flushed by background flushing or `fsync`. The metadata consistency is guaranteed by the FS-FLAC collaboration logging (§4.3).

Inspired by SCMFS [50] and ctFS [31], FlacFS allocates consecutive virtual memory addresses on the FLAC space for each file to store data. File inode only needs to record the start virtual address and the file size. FlacFS uses a buddy-like allocator for it. When the file size increases, a new range of consecutive virtual addresses is allocated, then the pages (existing and new) are attached to the new virtual addresses, and finally the old virtual addresses are reclaimed. This design allows FlacFS to leverage MMU to accelerate page indexing without the need for complex index structures (*e.g.*, B-tree).

4.2 Data Management

The data area is run on top of FLAC, which is created by `init_flac`. It appears to FlacFS as a range of consecutive kernel virtual memory addresses, and the data I/Os on the FLAC space are transparently cached.

File Read/Write. After the file is successfully opened, FlacFS calculates the target address range on FLAC space of the request by the start virtual address of the file (recorded in the inode) and the offset. Read and write are executed by `zcopy_from_flac` and `zcopy_to_flac` respectively, which makes the data transferring between file system and application is zero-copy.

Background Flushing. FlacFS launches a background thread periodically (10ms by default) to traverse the opened files and flush the dirty pages and related metadata to PM. It uses the 2-Phase flushing mechanism of FLAC for efficient data synchronization. For each dirty file, FlacFS creates a flush handle and collects the dirty pages according to the per-file dirty bitmap, and then uses `pflush_add` to add them to the handle (*i.e.*, attach to a temporary flush buffer). After collecting, FlacFS calls `pflush_commit` to atomically persist the dirty data to PM.

File Synchronization. Similar to traditional file systems, FlacFS provides `fsync` for users to flush data from DRAM to PM immediately. FlacFS uses the 2-Phase flushing mechanism to synchronize dirty data in `fsync`, which is similar to the background flushing. Following the semantics of `fsync`, the operation is returned after the data is persisted.

4.3 Security and Consistency

Data is protected by the kernel mode. FLAC is implemented in the kernel, and userspace applications can access it only through syscall/ioctl. Pages are always mapped to the application as read-only, which ensures that local operations of the application do not affect the data in the cache and other applications as they are handled by COW page fault. The metadata security can be solved by using the userspace security mechanisms or putting metadata management in the kernel. For example, the mechanism of existing systems [31, 57] can be used to ensure the metadata security, *i.e.*, the metadata area is protected by MPK [13, 42] and access permission only is granted to the user process during the metadata operation.

The FS-FLAC collaboration logging mechanism requires the upper-layer file system to provide formatted metadata modification and corresponding metadata recovery functions. FlacFS uses the newest inode as the *fs_metalog* parameter in persistent data modification APIs (`pflush_commit`, `pfree`), and overloads the external recovery function to overwrite the original inode by its newest version. FlacFS calls `init_flac` to recover the FLAC space when system restarts. After the success of `init_flac`, FlacFS rebuilds the metadata area in DRAM and the system is recovered from the crash.

4.4 Advantages of FlacFS/FLAC

FLAC allows file systems based on it to benefit from the DRAM cache while reducing the effects of "cache tax" as much as possible. Table 2 gives a comparison between FlacFS/FLAC and existing systems.

vs. Cache-based File Systems/mmap. There are many file systems designed based on the VFS. Although the VFS page cache can improve the performance in some scenarios in heterogeneous memory file systems, these systems suffer from heavy "cache tax" and fail to optimize the application-storage data transfer. These file systems also provide the mmap method to avoid the data transfer overhead, but it makes application design and storage backend to be coupled. Therefore, they cannot fully exploit the potential of cache in heterogeneous memory architecture.

vs. DAX-based File Systems. DAX-based systems bypass the DRAM cache in data I/O, making them suffer from high application-storage transfer overhead. Also, the latency and

Table 2: Comparison with Related Work

Type	Typical System	Data Cache	Low/Non Cache Tax Impact	App-Storage Zero-Copy	App-Storage Decouple
Cache-based FS	VFS page cache FSes (*e.g.*, ETX4, XFS [44], SPFS [49])	✔	✗	✗	✔
Cache-based mmap	mmap in VFS page cache FSes (*e.g.*, EXT4, XFS [44])	✔	✗	✔	✗
DAX-based FS	NOVA [51], SplitFS [20], WineFS [19], ctFS [31], KucoFS [5], PMFS [10] libnvmmio [6], EXT4-DAX [7], HTMFS [57], OdinFS [62], ZoFS [8]	✗	✔	✗	✔
DAX-based Runtime	Twizzler [4], Mnemosyne [46], PMDK [15] zIO [43], DaxVM [1], SubZero [22]	✗	✔	✔	✗
Flat Cache	FlacFS	✔	✔	✔	✔

concurrency of PM hardware greatly limit their performance. In particular, some DAX-based file systems also use remapping: SplitFS [20] proposes `relink`, an operation to atomically move a contiguous extent from one file to another, which is used to accelerate appends and atomic data operations; ctFS [31] proposes `pswap` to swap the page mapping of two same-sized contiguous virtual addresses, which is used to reduce the overhead of maintaining file data in contiguous virtual addresses. However, neither SplitFS nor ctFS uses remapping to optimize data copying between applications and file systems, and FLAC optimizes this part with the zero-copy caching technique. Some DAX-based systems focus on special design objectives, such as NUMA optimization (*e.g.*, OdinFS [62]), userspace optimization (*e.g.*, KucoFS [5], ZoFS [8], Trio [61]), and aging problem (*e.g.*, WineFS [19]). They are complementary to FlacFS.

vs. DAX-based Runtime. This type of work usually provides a memory management library or programming framework for applications. Although the overhead of data transfer between the application and storage system can be avoided, they require the application to be co-designed with the storage backend (*e.g.*, use customized interfaces or object abstraction). Some of these works provide zero-copy PM I/O libraries [22, 43]. However, they require applications to allocate read/write buffers on PM to avoid data copy, and thus force to ship the data processing from DRAM to PM, which is not friendly for some cases [48]. DAX-based runtime focuses on programming directly on PM and can be seen as complementary to the file system.

vs. Other Related Work. Some PM-based file systems try to use DRAM as a cache (*e.g.*, HiNFS [37] and HasFS [32]). However, these works do not exploit the potential of the virtual memory subsystem in the cache and are designed for the simulated PM. Some file systems (*e.g.*, Strata [24], Ziggurat [60]) are optimized for other multi-layer storage architectures (DRAM-PM-SSD). Some work focuses on data management in tiered memory (*e.g.*, HeMem [39] and Johnny Cache [29]), which are complementary to FLAC.

5 Discussion

Although FLAC/FlacFS offers promising performance, it also encounters some new challenges, which we discuss below.

Page Fault Overhead. COW page fault doesn't happen at every write, and is only triggered at the first time to overwrite the buffer. The natural COW page fault overhead is high and our evaluation shows that it can reduce performance by about 30 times in the worst case without specific optimization. Our optimizations (`bfault/detach`) precisely address two key bottlenecks in COW page fault and they are easy to adapt to applications (shown below). According to our evaluation, `bfault/detach` can reduce more than 78.3% COW page fault overhead in the worst cases (§ 6.2.4) and can be used effectively in real-world scenarios (§ 6.3).

Application Adaptation. We think adaptation is simple and straightforward: First, it requires only a few code changes. We intercepted the POSIX interface to transparently adopt the file operations (open, read, write, *etc*) to FlacFS. The code changes are related only to buffer allocation and page fault optimization. Second, it needs no change to the original application code logic. The code changes are alternative (replacing buffer allocation) and/or incremental (adding `bfault/detach` before reusing the buffer). This allows applications to be "trivially" adapted to FLAC/FlacFS.

Target I/O Workloads. FLAC/FlacFS is more friendly to large I/Os, especially I/Os larger than 64KB (§ 6.2.2). Large I/Os are important in production scenarios. For example, LLM (Large Language Model) training usually makes checkpoints in the file system for recovery. Take GPT3-NEOX [12] as an example, the average I/O size generated during checkpointing is at MB-level; As another example, SQL databases (*e.g.*, openGauss [36]) typically aggregate data into large blocks (*e.g.*, 64KB) and write to the file system by large I/Os.

Design Universality. Although this work mainly focuses on the cache framework of file systems, FLAC is possible to be adapted to other storage systems. For example, KV stores (*e.g.*, [2, 3, 21, 27, 28, 47, 58, 59]) can build their DRAM cache upon the FLAC space to enjoy the benefits of zero-copy caching and efficient cache management.

6 Evaluation

We compare FlacFS to a wide range of heterogeneous memory file systems to demonstrate the benefits of FLAC framework.

Cache-based Systems. Systems of this type include EXT4 and FlacFS. EXT4 is representative of file systems using the

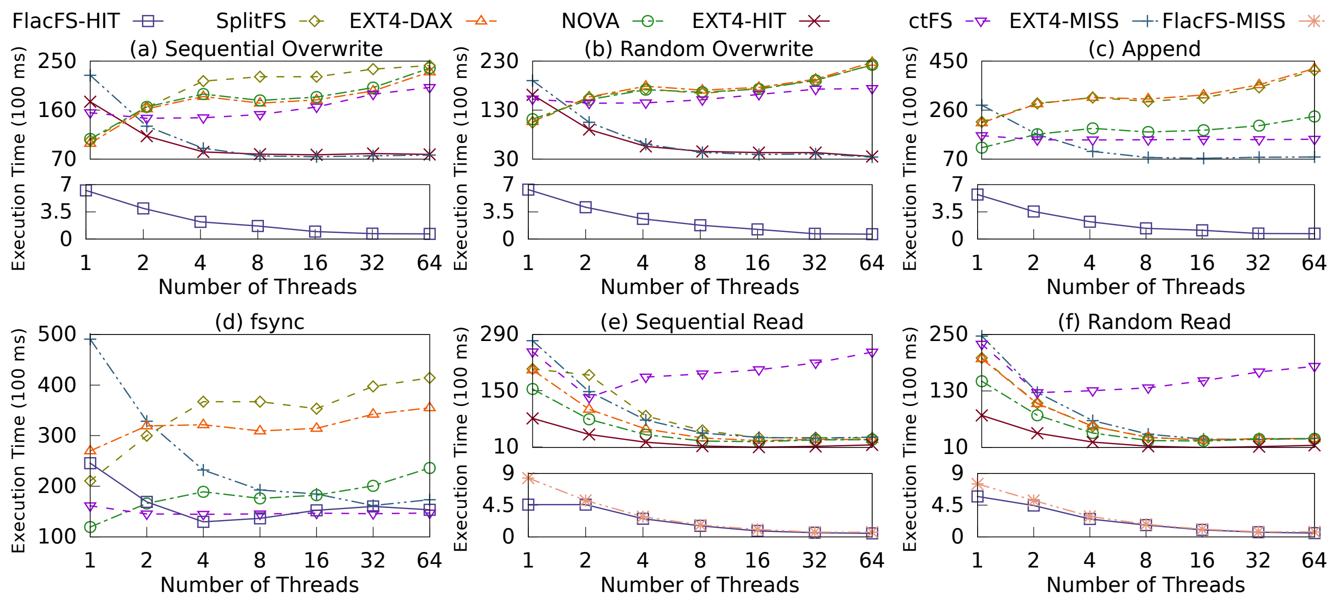


Figure 7: Micro Benchmark Performance.

VFS page cache (*e.g.*, XFS [44] and SPFS [49]). The dirty data flushing period is set to 10ms and 100ms for FlacFS and EXT4, respectively. FlacFS ensures the consistency of metadata and data, while EXT4 only ensures metadata consistency (ordered mode). If not specified, EXT4 and FlacFS trigger page eviction unless memory allocation fails, which is the default policy in Linux.

DAX-based Systems. Systems of this type includes EXT4-DAX [7], NOVA [51], SpiltFS [20], and ctFS [31]. Data I/Os of these systems bypass the VFS page cache and perform on PM directly. NOVA is set to sync mode, while SplitFS and ctFS are set to POSIX mode. All tested DAX file systems only ensure the metadata consistency, while FLAC ensures both metadata and data consistency.

Testbed. All experiments are run on a server with two Intel Xeon CPUs, 256GB RAM, and 1TB (128GB$\times$8) PM. FlacFS and EXT4 use Ubuntu 20.04 with Linux 5.1, and others file systems use the kernel versions they can support.

6.1 Benchmark Performance

6.1.1 Micro Benchmark

We evaluate the duration of performing append, overwrite, read, and fsync-after-append (`fsync` is called after each write) on 64 1GB files with random and sequence patterns. The I/O size is 2MB and there is no contention for accesses between files in these experiments. Figure 7 shows the results. For the cache-based file systems, "*-HIT" and "*-MISS" represent cache hits and misses, respectively (analyzed in §6.2.1).

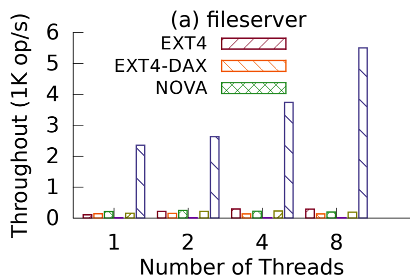
In the write scenarios, FlacFS provides a maximum performance increase of more than two orders of magnitude over other tested systems. In the read scenarios, FlacFS outperforms other tested systems by more than 200 times. The zero-copy caching in FLAC significantly reduces the data copy overhead between the application's write buffer and the file system, while all other systems suffer from this copying overhead. Compared with another cache-based system, EXT4, the data persisting phase during background flushing in FlacFS does not block the front-end writes, which significantly improves the performance in write-intensive scenarios. In the fsync-after-append scenario, FlacFS is comparable to the best of the DAX file systems and better than EXT4. Although dense `fsync` is not friendly to FlacFS, it still performs well due to the lightweight nature of FLAC.

At the framework level, we observe that the DAX-based systems have lower scalability than cache-based systems (EXT4 and FlacFS) under write-intensive workloads. The DAX approaches are difficult to scale beyond even 2 concurrent threads in Figure 7 (a) - (d) because they reach the bandwidth and concurrency limitation of PM. In summary, these results demonstrate that FLAC can fully exploit the potential of DRAM cache in heterogeneous memory file systems.

6.1.2 Macro Benchmark

We use two I/O intensive workloads in Filebench [45] to evaluate the performance of FlacFS in the scenarios with mixed operations (including many types of data and metadata operations). All workloads use 128MB file and 2MB I/O. The main process of Filserver is to create files, write data to the files, and then read data from the files. The main process of Webserver is to create and append files, and then read the files repeatedly. In particular, read operations have stronger locality than write operations in these workloads.

Figure 8 shows that the throughput of FlacFS is higher than other tested file systems by more than 40 times and 20 times in Fileserver and Webserver, respectively. At the same



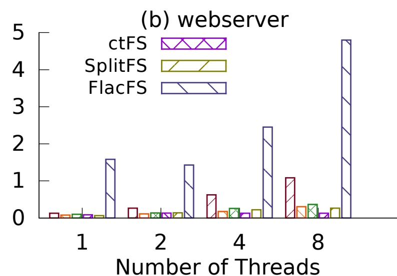


Figure 8: Filebench Performance.

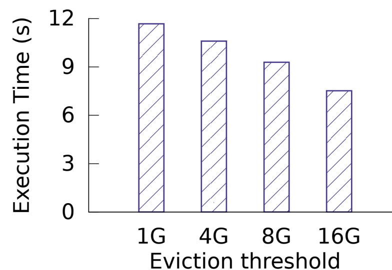
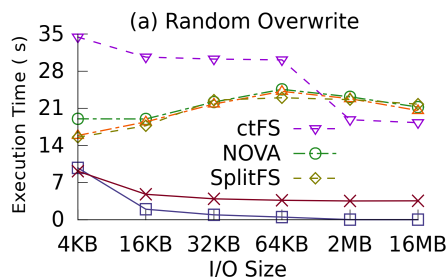
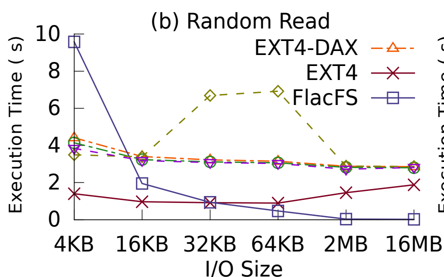


Figure 9: Cache Eviction Overhead.





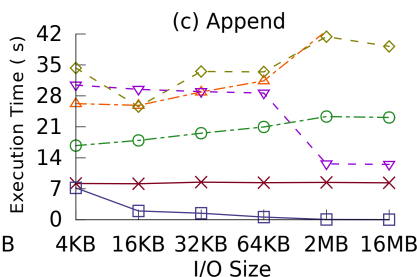


Figure 10: Impact of I/O Size.

time, the concurrency of FlacFS is better than other tested systems. The DAX-based systems are limited by the hardware disadvantages of PM in these experiments. The other cache-based file system, EXT4, is also better than the DAX-based systems because of the locality of the workload, especially in the Webserver case. However, EXT4's performance is still significantly lower than FlacFS because of the inefficiency of the VFS page cache framework.

6.2 Design Analysis

6.2.1 Impact of DRAM Cache Size

Cache size affects the overall performance through two aspects: overhead of cache miss and page eviction.

Cache Miss Overhead. The hit ratio is determined by the cache policy and the workload behavior. As this work mainly focuses on the cache framework design, we just show the performance of the upper (100% hit) and lower (100% miss) bounds. We clear the DRAM cache before each run to evaluate the system performance under cache miss. In particular, write operations in FlacFS do not encounter cache misses in these experiments because the new pages are always attached from the application's DRAM buffer to the FLAC space. By comparing the "EXT4-MISS" and "FlacFS-MISS" in Figure 7 (e) and (f), we found that FlacFS outperforms EXT4 by more than 320 times, which benefits from the asynchronous cache miss handling mechanism. In FLAC, the heterogeneous memory addressing allows pages to be accessed directly whether it is in DRAM or PM, so uncached pages can be attached to the application's read buffer and loaded to DRAM in the background. Therefore, the latency penalty of cache miss is hidden for front-end data I/Os. This design also allows FlacFS to perform better than DAX-based file systems in the cache miss scenario because they need to synchronously copy data from PM to the application buffers.

Eviction Overhead. We append 16GB of data to the files with different eviction thresholds. This experiment is used to measure pure eviction overhead because appending does not have data locality. A smaller threshold means a smaller effective cache capacity and causes more data to be evicted and higher eviction frequency. For example, with 16G threshold, no data is evicted, while half of the data (8G) is evicted at once when the threshold is 8G. The major overhead in eviction comes from copying pages to PM and its performance is bounded by the PM bandwidth. The eviction involves the extra overhead of updating page table entries and invalidating TLB. Figure 9 shows the eviction performance. As expected, a smaller threshold introduces more penalties. For instance, eviction cost under 1G threshold is 2.3 times of 8G threshold, as 1G threshold has nearly twice the amount of eviction data as 8G threshold (15G *vs.* 8G). Additionally, 1G threshold causes more TLB invalidation overhead due to more frequent eviction than 8G threshold.

To sum up the above experiments, we believe that with efficient cache algorithms (out of the scope of this work), FLAC can run efficiently in cache-starved scenarios.

6.2.2 Impact of I/O Size

We evaluate the duration of performing random read/overwrite and append in the I/O sizes ranging from 4KB to 16MB with 64 concurrent threads (no contention). Figure 10 shows that FlacFS has significant advantages compared to other systems when the I/O size is greater than 64KB, because the data copy and migration are the major overheads in these scenarios and this meets the optimization point of FlacFS. For I/Os smaller than 64KB, the advantage of FlacFS decreases as the

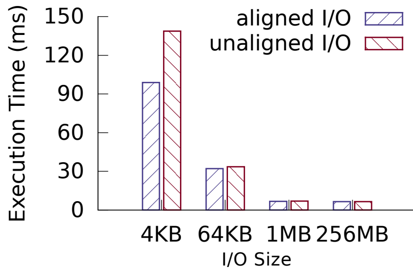


Figure 11: Impact of Unaligned Page.

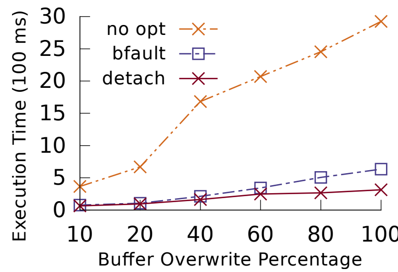


Figure 12: Impact of COW Page Fault.

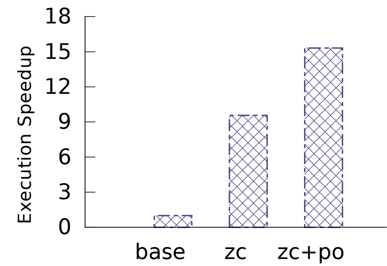


Figure 13: Performance Breakdown.

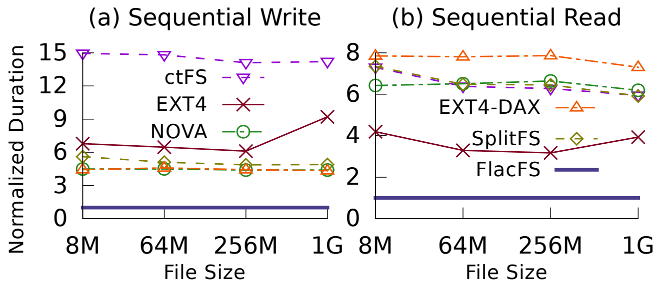


Figure 14: Impact of File Size. For each file size, the duration of other file systems are normalized by FlacFS.

I/O size decreases, as the additional overhead introduced by FlacFS (*e.g.*, TLB flush) becomes apparent in these scenarios. As discussed in § 5, we believe the FlacFS-friendly scenarios can cover a lot of practical workloads. In scenarios where the I/O size is smaller than a page (4KB), they are generally not file system friendly because file systems manage data at a page granularity. Therefore, many real-world applications try to avoid triggering file I/Os smaller than 4KB.

6.2.3 Impact of Page Alignment

FLAC can serve file I/O at any offset and size. The automatic alignment and sliding window buffer are used to solve the page unaligned problem. We evaluate the impact of page unaligned on performance by randomly overwriting 1GB of data in the file under different I/O sizes. Figure 11 shows that unaligned I/Os have a performance degradation of about 20% compared to aligned I/Os when the I/O size is 4KB. However, unaligned accesses have little impact on performance as the I/O size increases, because the amount of data copied by the sliding window buffer does not exceed 4KB, so the proportion of this overhead decreases with the increase in I/O size.

6.2.4 Impact of COW Page Fault

Pages in the application buffer are set to read-only when they are attached to/from FLAC for security and isolation. As a result, COW page faults are triggered when the application updates the data in the buffer for the first time after the FlacFS read/write. FLAC proposes two APIs for batch faulting (`bfault`) and detaching (`detach`) for applications to reduce or eliminate the negative effect of COW page faults. We use 16 test threads to random write on 16 files and rewrite the data in the buffer by using `memset` after each write to evaluate these optimizations (read scenario exhibits a similar performance pattern). Figure 12 shows the results. As the baseline ("no opt"), the test threads simply rewrite the read/write buffers so that normal COW page faults will be triggered. Relatively, the test threads call `bfault` or `detach` for the buffer after each FlacFS read/write to show the benefits of batch faulting and detaching.

The results shows that the total execution time of the baseline grows significantly as the percentage of buffer overwrites increases, because the higher the overwrite percentage, the more COW page faults are triggered, which results in the increased overhead of TLB flushing and data copy. In comparison, batch faulting and detaching can reduce the total execution time by 78.3% and 89.2%, respectively, when the overwrite percentage reaches 100%. Batch faulting reduces the overhead of TLB flushing by aggregating multiple COW page faults. Further, detaching completely avoids COW page faults by remapping new pages to the given addresses.

6.2.5 Performance Breakdown

FLAC includes the key techniques of zero-copy caching and parallel-optimized cache management. As the baseline, we implement a simple FLAC equipped with only a heterogeneous page table and use memory copy to transfer data between the FLAC space and the application buffer. Therefore, both zero-copy caching and parallel optimizations are removed from this simple FLAC.

We use 2 concurrent threads to perform 2MB random write I/Os in this experiment. Figure 13 shows the performance breakdown. In the "zc" case, we add the zero-copy caching design into the baseline but use the coarse-grained lock instead of the 2-Phase flushing (*i.e.*, the front-end I/Os are blocked during the data synchronization). The results show that the performance can be improved by around 10 times by adding the zero-copy caching. In the "zc+po" case, both zero-copy caching and parallel optimizations are applied, and the performance is improved by about 15 times compared to the baseline. For the parallel optimizations, this experiment focus on the contribution of 2-Phase flushing, while the benefits of asynchronous cache miss handling are reflected in §6.2.1.

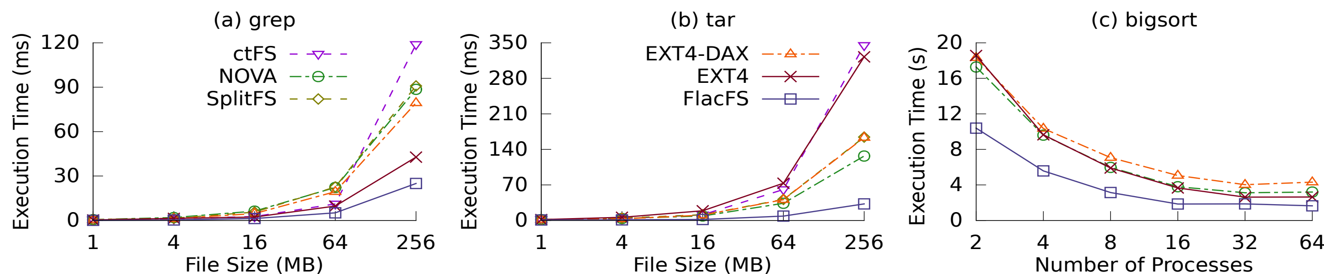


Figure 15: Performance on Real-World Applications.

This experiment shows that FLAC addresses the important bottlenecks of the heterogeneous memory cache framework.

6.2.6 Impact of File Size

We perform 64KB I/Os on different sizes of files (8MB to 1GB) with a single thread to show the impact of file size on the advantages of FLAC. We normalize the duration of other file systems to FlacFS to reflect the fluctuation of FlacFS' performance improvement, *i.e.*, the smoother curve indicates that the file size has less impact on performance improvement. Figure 14 shows that FlacFS has a smooth performance advantage under different file sizes: it has a third of the duration of the second-best system in write and read. The reason is that the performance improvement of FlacFS mainly comes from the zero-copy and parallel optimization, which are not strongly related to the file size.

6.3 Real-World Applications

We evaluate FlacFS in some real-world applications to demonstrate its end-to-end performance benefits. For each application, we replace the file system calls and buffer allocation by the FlacFS' interfaces and sliding window buffer mechanism to port it to our system. In addition, we use batch fault or detach (select based on how the buffer is used) for the read/write buffer to optimize the COW page fault overhead before the application reuses the buffer (if have).

6.3.1 Command Line Application

We port two widely used command line utilities to use FlacFS. The first one is `grep` v3.7. We measure the execution time of matching a character within the input file. Figure 15 (a) shows the performance of increasing file size The `grep` only issues read operations and FlacFS runs 6.7 times faster than the best DAX file system (ctFS) and 4.8 times faster than EXT4 at 1MB file size. The second application is `tar` v1.34. The `tar` contains not only read operations but also contains write to generate the output archive. Figure 15 (b) plots the execution time of creating an archive from the input file. FlacFS still achieves the best performance. With 16MB file, FlacFS gets 4.4 times improvement over the best DAX file system (NOVA) and 9.4 times better than EXT4. Additionally, the computation in `tar` is less expensive than `grep`, which process regulator expression matching. Thus, `tar` spends more time on file I/Os than `grep` and the performance gain in `tar` is more than `grep`. For instance, with 256MB file, FlacFS improves over SplitFS by 3.6 and 5.6 times for `grep` and `tar`, respectively.

6.3.2 Big Data Processing

We evaluate FlacFS and other file systems with BigSort [25], a large-scale merge sort application implemented by Lawrence Livermore National Laboratory. Merge sort is an important phase in big data processing (*e.g.*, page ranking). Given a dataset, BigSort partitions it and performs the merge sorted on each partition recursively. There are three phases in each merge sort: Phase 1) reads the unsorted objects from the target file; Phase 2) performs quick sorting on the objects read in the previous phase; Phase 3) stores the intermediate-ordered results in the file system. After the recursive exit, the global ordered results are written to the output file.

We perform merge sorting on a dataset of 134 million integers. Porting BigSort to SplitFS and ctFS causes multiple processes to hang, so we cannot obtain their performance results. Figure 15 (c) shows that FlacFS has up to 2.62 times improvement compared to other file systems when the number of concurrent processes reaches 64. Benefiting from the zero-copy caching design, FlacFS has a significant performance advantage in Phases 1 and 3 because they include intensive large file I/Os (512KB per I/O). Phase 2 is compute-intensive, and it will incur an unnegligible overhead of COW page fault if nothing is done to optimize it. As a result, FlacFS has an obvious performance advantage in this complex application.

7 Conclusion

Heterogeneous memory provides various advantages, but it also poses challenges to the file system architecture. We analyze the shortcomings of existing cache-based and DAX-based storage frameworks in heterogeneous memory, and conclude that DRAM cache still has great potential in fast all-memory architectures. We propose FLAC, a flat cache framework of heterogeneous memory that integrates the page cache into the virtual memory subsystem. FLAC unlocks the potential of cache through zero-copy caching and parallel-optimized cache management. We implement a file system based on FLAC and show that FLAC has significantly better performance than existing cache and DAX solutions.

Acknowledgments

We thank our shepherd Oana Balmau and the anonymous reviewers for their constructive comments and feedback. We also thank our colleagues in the Huawei OS Kernel Lab for their support. Yuxin Ren is the corresponding author.

References

[1] Chloe Alverti, Vasileios Karakostas, Nikhita Kunati, Georgios Goumas, and Michael Swift. DaxVM: Stressing the limits of memory as a file interface. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'22)*, pages 369–387, 2022.

[2] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *Proceedings of the USENIX Technical Conference (ATC'17)*, pages 363–375, 2017.

[3] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing latency spikes in log-structured merge key-value stores. In *Proceedings of the USENIX Technical Conference (ATC'19)*, pages 753–766, 2019.

[4] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller. Twizzler: A data-centric os for non-volatile memory. In *Proceedings of the USENIX Annual Technical Conference (ATC'20)*, pages 1–31, 2020.

[5] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. Scalable persistent memory file system with kernel-userspace collaboration. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'21)*, pages 81–95, 2021.

[6] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. Libnvmmio: Reconstructing software IO path with failure-atomic memory-mapped interface. In *Proceedings of the USENIX Annual Technical Conference (ATC'20)*, pages 1–16, 2020.

[7] Johnathan Corbet. Ext4-dax. https://lwn.net/Articles/717953, October 2022.

[8] 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'19)*, pages 478–493, 2019.

[9] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'93)*, pages 189–202, 1993.

[10] 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 European Conference on Computer Systems (EuroSys'14)*, pages 1–15, 2014.

[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 European Conference on Computer Systems (EuroSys'16)*, pages 1–16, 2016.

[12] EleutherAI. GPT3-NEOX. https://github.com/EleutherAI/gpt-neox, October 2023.

[13] 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 *Proceedings of the USENIX Annual Technical Conference (ATC'19)*, pages 489–504, 2019.

[14] Intel. 3D XPoint DCPMM. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory, September 2021.

[15] Intel. Persistent memory development kit. https://pmem.io/pmdk, October 2022.

[16] 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. *arXiv:1903.05714*, 2019.

[17] Song Jian and Xiaodong Zhan. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the ACM Sigmetrics Conference (SIGMETRICS'02)*, 2002.

[18] Myoungsoo Jung. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *Proceedings of the ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22)*, pages 45–51, 2022.

[19] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. WineFS: A

hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'21)*, pages 804–818, 2021.

[20] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'19)*, pages 494–508, 2019.

[21] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. SLM-DB: single-level key-value store with persistent memory. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'19)*, pages 191–205, 2019.

[22] Juno Kim, Yun Joon Soh, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson. SubZero: Zero-copy IO for persistent main memory file systems. In *Proceedings of the Asia-Pacific Workshop on Systems (APSys'20)*, pages 1–8, 2020.

[23] Miryeong Kwon, Sangwon Lee, and Myoungsoo Jung. Cache in hand: Expander-driven CXL prefetcher for next generation CXL-SSDs. In *Proceedings of the ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'23)*, pages 24–30, 2023.

[24] 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'17)*, pages 460–477, 2017.

[25] Lawrence Livermore National Laboratory. Bigsort. https://gitlab.com/arm-hpc/benchmarks/coral-2/BigSort, May 2023.

[26] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-change technology and the future of main memory. *IEEE Micro*, 30(1):143–143, 2010.

[27] 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 ACM Symposium on Operating Systems Principles (SOSP'19)*, pages 447–461, 2019.

[28] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell+: Snapshot isolation without snapshots. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 425–441, 2020.

[29] Baptiste Lepers and Willy Zwaenepoel. Johnny Cache: the end of dram cache conflicts (in tiered main memory systems). In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI'23)*, pages 519–534, 2023.

[30] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. SocksDirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*, pages 90–103. 2019.

[31] Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, and Ding Yuan. ctFS: Replacing file indexing with hardware memory translation through contiguous file allocation for persistent memory. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'22)*, pages 35–50, 2022.

[32] Yubo Liu, Hongbo Li, Yutong Lu, Zhiguang Chen, Nong Xiao, and Ming Zhao. HasFS: optimizing file system consistency mechanism on nvm-based hybrid storage architecture. *Cluster Computing*, 23:2510–2515, 2020.

[33] Yubo Liu, Yuxin Ren, Mingrui Liu, Hanjun Guo, Xie Miao, and Xinwei Hu. Cache or direct access? revitalizing cache in heterogeneous memory file system. In *Proceedings of the Workshop on Disruptive Memory Systems (DIMES'23)*, pages 38–44, 2023.

[34] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit O. Kanaujia, and Prakash Chauhan. TPP: transparent page placement for CXL-enabled tiered memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*, pages 742–755, 2023.

[35] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'03)*, pages 115–130, 2003.

[36] openGauss. openGauss. https://opengauss.org/, December 2023.

[37] 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'16)*, pages 1–16, 2016.

[38] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI'99)*, 1999.

[39] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'21)*, pages 392–407, 2021.

[40] Yuxin Ren, Gabriel Parmer, Teo Georgiev, and Gedare Bloom. CBufs: Efficient, system-wide memory management and sharing. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM'16)*, pages 68–77, 2016.

[41] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'21)*, pages 341–354, 2021.

[42] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. CubicleOS: A library os with software componentisation for practical isolation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, pages 546–558, 2021.

[43] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. zIO: Accelerating IO-Intensive applications with transparent Zero-Copy IO. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI'22)*, pages 431–445, 2022.

[44] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proceedings of the USENIX Technical Conference (ATC'96)*, pages 363–375, 1996.

[45] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX ;login:*, 41(1):6–12, 2016.

[46] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, pages 91–104, 2011.

[47] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. Pacman: An efficient compaction approach for log-structured key-value store on persistent memory. In *Proceedings of the USENIX Technical Conference (ATC'22)*, pages 773–788, 2022.

[48] Yongfeng Wang, Yinjin Fu, Yubo Liu, Zhiguang Chen, and Nong Xiao. Characterizing and optimizing hybrid DRAM-PM main memory system with application awareness. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'22)*, pages 879–884, 2022.

[49] Hobin Woo, Daegyu Han, Seungjoon Ha, Sam H. Noh, and Beomseok Nam. On stacking a persistent memory file system on legacy file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'23)*, pages 281–296, 2023.

[50] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, pages 1–11, 2011.

[51] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'16)*, pages 323–338, 2016.

[52] Jian Yang, Juno Kim, Morteexe 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'20)*, pages 169–182, 2020.

[53] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. GL-Cache: Group-level learning for efficient and high-performance caching. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'23)*, pages 115–133, 2023.

[54] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and K. V. Rashmi. FIFO queues are all you need for cache eviction. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'23)*, pages 130–149, 2023.

[55] Qirui Yang, Runyu Jin, Bridget Davis, Devasena Inupakutika, and Ming Zhao. Performance evaluation on CXL-enabled hybrid memory pool. In *Proceedings of the International Conference on Networking, Architecture and Storage (NAS'22)*, pages 1–5, 2022.

[56] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. Overcoming the memory wall with CXL-enabled SSDs. In *Proceedings of the USENIX Technical Conference (ATC'23)*, pages 601–617, 2023.

[57] Jifei Yi, Mingkai Dong, Fangnuo Wu, and Haibo Chen. HTMFS: Strong consistency comes for free with hardware transactional memory in persistent memory file systems. In *Proceedings of the USENIX Conference on*

File and Storage Technologies (FAST'22), pages 17–34, 2022.

[58] Baoquan Zhang and David HC Du. NVLSM: A persistent memory key-value store using log-structured merge tree with accumulative compaction. *ACM Transactions on Storage*, 17(3):1–26, 2021.

[59] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. ChameleonDB: a key-value store for Optane persistent memory. In *Proceedings of the European Conference on Computer Systems (EuroSys'21)*, pages 194–209, 2021.

[60] 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'20)*, pages 207–219, 2020.

[61] Diyu Zhou, Vojtech Aschenbrenner, Tao Lyu, Jian Zhang, Sudarsun Kannan, and Sanidhya Kashyap. Enabling high-performance and secure userspace nvm file systems with the trio architecture. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'23)*, pages 150–165, 2023.

[62] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. ODINFS: Scaling PM performance with opportunistic delegation. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, pages 179–193, 2022.

Kosmo: Efficient Online Miss Ratio Curve Generation for Eviction Policy Evaluation

Kia Shakiba, Sari Sultan, and Michael Stumm
University of Toronto

Abstract

In-memory caches play an important role in reducing the load on backend storage servers for many workloads. Miss ratio curves (MRCs) are an important tool for configuring these caches with respect to cache size and eviction policy. MRCs provide insight into the trade-off between cache size (and thus costs) and miss ratio for a specific eviction policy. Over the years, many MRC-generation algorithms have been developed. However, to date, only Miniature Simulations is capable of efficiently generating MRCs for popular eviction policies, such as *Least Frequently Used* (LFU), *First-In-First-Out* (FIFO), 2Q, and *Least Recently/Frequently Used* (LRFU), that do not adhere to the inclusion property. One critical downside of Miniature Simulations is that it incurs significant memory overhead, precluding its use for online cache analysis at runtime in many cases.

In this paper, we introduce Kosmo, an MRC generation algorithm that allows for the simultaneous generation of MRCs for a variety of eviction policies that do not adhere to the inclusion property. We evaluate Kosmo using 52 publicly-accessible cache access traces with a total of roughly 126 billion accesses. Compared to Miniature Simulations configured with 100 simulated caches, Kosmo has lower memory overhead by a factor of 3.6 on average, and as high as 36, and a higher throughput by a factor of 1.3 making it far more suitable for online MRC generation.

1 Introduction

In-memory caches play an important role in reducing the load on backend storage servers for many workloads [1–6]. These caches improve scalability and can reduce the latency of data access requests by serving data directly from main memory. Redis [7] and Memcached [8] are two popular in-memory caches, both of which are open source and often provided as a service by cloud providers [9–12].

In-memory caches can consume a large portion of a data center's operating budget, sometimes exceeding 60% of the total operating cost [13]. In cloud-hosted environments, such caches are priced proportionately to their size. As such, it is important to provision each cache to the "right" size using the cost-performance trade-offs for its workloads: caches that are too small incur higher miss ratios and thus higher backend storage server loads, while caches that are too large consume unnecessary resources and have higher operational costs.

One of the most effective tools to understand the trade-off between cache size and miss ratio is the *miss ratio curve* (MRC), and over the years, many MRC-generation algorithms have been developed [14–22]. An MRC plots a cache's miss ratio as a function of the cache size. Figure 1 depicts an example of such an MRC. The MRC shows the effect on the miss ratio of varying the cache's size from 0GiB to 400GiB under the MSR `src1` workload [23] using the *Least Frequently Used* (LFU) eviction policy. There is a sudden drop in the miss ratio between roughly 160GiB and 190GiB. Such a drop is referred to as a cliff and knowledge of its presence is particularly useful: if the cache were initially configured with 160GiB of memory, the MRC indicates that increasing the cache size by 30GiB would result in roughly 30% improvement in the miss ratio. The plateaus between 70GiB and 160GiB, and 190GiB and 270GiB are also informative: they indicate that if the cache is currently configured to a size within one of the plateaus, then the size can be decreased to 70GiB or 190GiB, respectively, without severely impacting the miss ratio.

The choice of eviction policy is also an important factor in configuring an in-memory cache. While most in-memory caches default to the *Least Recently Used* (LRU) eviction policy, it has been shown that under certain workloads, caches operate more efficiently using non-LRU eviction policies [3, 24–27]. For example, the LFU eviction policy can sometimes achieve a roughly 14% reduction in miss ratio when allocated the same cache size and under the same workload [27]. Eviction policies such as *First-In-First-Out* (FIFO) also have lower computational and memory overheads than other policies, such as LRU [3].

To optimize a cache configuration in terms of both size and eviction policy, it is necessary to generate an MRC for

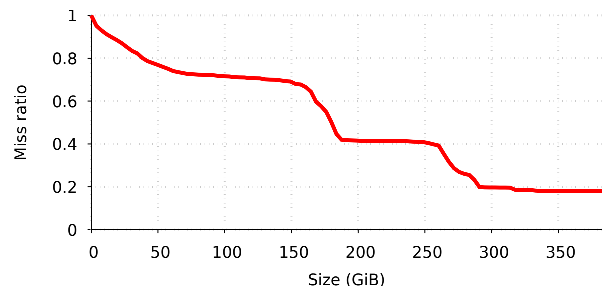


Figure 1: *MRC generated for MSR `src1` workload [23] using the Least Frequently Used (LFU) eviction policy.*

each eviction policy under consideration. However, a key limitation of almost all existing MRC-generation algorithms is that they only model caches operating with an eviction policy that satisfies the inclusion property (e.g., LRU); as such, they do not support eviction policies such as LFU, FIFO, 2Q, *Least Recently/Frequently Used* (LRFU), or *Most Recently Used* (MRU). The only known MRC generation algorithm capable of modeling a wide array of non-LRU caches with reasonable computational efficiency is Miniature Simulations (*MiniSim*) [18]. It runs individual simulations of caches of different sizes and makes use of the SHARDS [17] sampling algorithm to improve its runtime performance. However, MiniSim has several serious drawbacks, the most notable being its high memory usage. MiniSim effectively simulates independent caches of varying sizes, often causing duplicate data to be stored in the internal structures of many of these simulated caches. We found through experimentation with numerous workloads that MiniSim configured with 100 simulated caches consumes the following amounts of memory on average: 113MiB for the LFU eviction policy, 57MiB for FIFO, 40MiB for 2Q, and 31MiB for LRFU, with up to 3.1GiB for the LFU eviction policy, 1.72GiB for FIFO, 396MiB for 2Q, and 597MiB for LRFU in extreme cases. Further, to generate MRCs for multiple eviction policies simultaneously, these memory requirements are compounded. With these memory requirements, MiniSim will likely consume substantial memory, and hence may even interfere with the cache itself.

This paper introduces **Kosmo**, an MRC generation algorithm that supports the simultaneous generation of MRCs for a variety of eviction policies while, on average, using significantly less memory than MiniSim, making it better suited for online MRC generation. Kosmo uses a novel method of calculating reuse distances through the introduction of *eviction maps*. We show how Kosmo can be used to simultaneously generate MRCs for six eviction policies: LFU, FIFO, 2Q, LRFU, LRU, and MRU. Notably, LFU, FIFO, 2Q, LRFU, and MRU do not adhere to the inclusion property (§2.2).

We evaluate Kosmo using a total of 52 publicly-available workloads and measure memory usage, throughput, and accuracy for LFU and FIFO, and 33 workloads for the 2Q and LRFU eviction policies. Kosmo requires an average of 3.6 times less memory, and up to 36 times less than MiniSim across all eviction policies. Kosmo has an average throughput 1.3 times that of MiniSim across all eviction policies. Finally, Kosmo, which is also an approximate generation algorithm, produces MRCs with comparable accuracy to those generated by MiniSim.

Contributions. The contributions we make in this paper are:

- We introduce Kosmo, a novel method of simultaneously generating MRCs for a variety of eviction policies.
- We introduce a method of reconstructing the stacks of caches of varying sizes using a single copy of the cached data through our novel data structure, *eviction maps*.
- We describe how to apply eviction maps to the LFU, FIFO, 2Q, LRFU, LRU, and MRU eviction policies, allowing Kosmo to generate MRCs for these policies.
- We evaluate the performance of both Kosmo and MiniSim and show that Kosmo achieves an average memory reduction of a factor of 3.6 and up to a factor of 36.
- We examine to what degree different eviction policies violate the inclusion property.

Limitations. The work we present has several limitations, however. First, we only describe Kosmo for six sample eviction policies. Although we know Kosmo supports additional eviction policies beyond those described in this paper, it remains an open problem which classes of eviction policies Kosmo is able to support. Second, the MRCs Kosmo generates are monotonically decreasing which could increase the error for eviction policies which display significant non-monotonic behaviour. Finally, we recognize that the performance of MiniSim is affected by the performance of the underlying cache it is simulating.

2 Background

In this section, we discuss relevant prior work. We first describe the eviction policies that are the focus of this paper, namely: LFU, FIFO, 2Q, LRFU, LRU, and MRU. We then discuss the inclusion property and its importance in MRC generation. Next, we describe several key MRC generation algorithms which provide the necessary background to understand the Kosmo algorithm. Mattson's algorithm gives insight into how MRC generation can be done for policies that do not violate the inclusion property (generally referred to as "stack-based eviction policies"). SHARDS is the sampling algorithm used by both Kosmo and MiniSim. MiniSim is currently the only known, reasonably computationally efficient method for generating MRCs for caches with non-stack-based eviction policies. It is the current state-of-the-art algorithm to which we compare Kosmo.

2.1 Eviction policies

LFU (Least Frequently Used) evicts the least frequently used object in the cache to make room for new objects. A simple method of implementing this policy is to use a stack of objects, ordered firstly by frequency count and secondly by last access time. If an object needs to be evicted, the one with the smallest frequency count, or oldest time on a tie, is selected. LFU caches often outperform LRU caches in workloads that exhibit a Zipfian distribution [24, 25].

FIFO (First-In-First-Out) evicts objects in the same order in which they first entered the cache. It can be implemented using a stack[1] of objects ordered by their entry times where the object with the oldest time is selected for eviction. This policy has been found to perform well on large workloads

[1]This is sometimes also referred to as a "queue" in this context. In this paper, however, we refer to the internal data structure which holds the objects of a cache as a "stack," regardless of eviction policy.

in which accesses have large inter-arrival gaps, such as those exhibited by scanning behaviours [3, 26]. Of all the eviction policies, it is the most efficient to implement [3].

LRU (Least Recently Used) and **MRU** (Most Recently Used) evict the least recently and most recently accessed object in the cache, respectively, to make room for a new object. To implement these policies, a stack of objects sorted by their last access times is used, where the object with the oldest (LRU) or youngest (MRU) time is evicted. LRU is perhaps the most widely-used eviction policy, though MRU has been found to perform better when the workload is cyclical [28].

2Q [29] maintains objects in a cache in two separate stacks: one for objects which have been accessed only once (the *A1* stack), and one for objects which have been accessed multiple times (the *Am* stack). Objects in the A1 stack are evicted in FIFO order, and objects in the Am stack are evicted in LRU order. The A1 stack is further partitioned into two stacks referred to as *A1in* and *A1out*[2] of size *Kin* and *Kout*, respectively, where Kin and Kout are ratios of the total cache size (the authors note that a Kin value of 25% and a Kout value of 50% work well in most cases). The A1in and A1out stacks differentiate themselves in the handling of objects that get accessed a second time; if the accessed object is in the A1out stack, it gets promoted to the Am stack, while if it is in the A1in stack, it does not. Upon the first access to an object, it is placed at the head of the A1in stack. If the A1in stack is full, the oldest object in the stack is removed and placed at the head of the A1out stack. If the A1out stack is full, the oldest object is evicted from the cache. If an object which already exists in the A1out stack is accessed, it is removed from the A1out stack and placed at the head of the Am stack. If the Am stack is full, an object is evicted using LRU.

LRFU (Least Recently/Frequently Used) combines objects' recency (i.e., the time since the object was last accessed) and frequency counts to determine which object to evict. Each object has an associated *Combined Recency and Frequency* (CRF) value computed as $CRF = \sum_{i=1}^{k} F(t_{now} - t_{access_i})$, where k is the number of times the object has been accessed previously, t_{now} is the current time, t_{access_i} is the time at which the object was accessed the ith time, and $F(x)$ is defined as $F(x) = (\frac{1}{p})^{\lambda * x}$, where p is a value greater than or equal to two, and λ is a value between 0 and 1. Tuning the value of λ allows the cache to behave more similarly to an LFU cache (with λ closer to 0) or an LRU cache (with λ closer to 1). The object with the smallest CRF value is selected for eviction. Although the described CRF formula requires the full history of the object's access times, the authors note that given an object's last access time and last CRF value, one can calculate the updated CRF value without needing the object's access history using $CRF_{updated} = F(0) + F(t_{now} - t_{last_access}) * CRF_{last}$. The LRFU policy has been shown to outperform many other policies for a number of important workloads [27].

[2] The A1out "ghost" stack holds references to objects, not their values.

2.2 Inclusion property

An important characteristic of an eviction policy is whether or not it adheres to the *inclusion property*. This property states that all objects that exist in a cache of size S at a given time, also exist in any cache of size $S' > S$, when given the same access trace [18, 30]. An extension of this property is the *strict inclusion property* which adds the further constraint that all common objects in any two caches (with the same access trace) must be in the same order in the caches' internal data structures (i.e., stacks).

An MRC generation algorithm that models an eviction policy adhering to the strict inclusion property is often referred to as a "stack algorithm," and can be implemented similarly to Mattson [14], described below. If the eviction policy does not adhere to the strict inclusion property, a dedicated algorithm for the eviction policy or MiniSim must be used.

In the literature, the LFU eviction policy is often referred to as a stack algorithm, which implies it can be modeled using an algorithm similar to Mattson [14, 17, 30]. However, this is only the case for so called *ideal LFU caches*, in which the cache maintains frequency counters for all objects that were ever accessed; if an object is evicted and accessed again in the future, its counter persists and is further incremented. In practice, LFU caches do not maintain the counters of evicted objects [31], as maintaining these counters would entail significant memory overhead. These *practical LFU cache* implementations remove the counter of any object being evicted from the stack, and if a previously evicted object is accessed again, a new counter is instantiated and initialized to one.

Practical LFU caches do not adhere to the inclusion property, in contrast to ideal LFU caches. Table 1 shows this for a simple access trace. Here, the objects and their associated counts in two practical LFU caches of size 3 and 4 are shown. At each time step, the frequency counter (shown alongside each object in brackets) of the accessed object is incremented by one, or initialized to one in the case of an object being accessed for the first time. The stack of each cache is ordered from least to most likely to be evicted from the cache (i.e., the object with the largest frequency counter, or most recently accessed if two objects have the same frequency counter, on the left). At time 9, it is evident that although the two caches were provided with the same access trace, object "e" exists in the cache of size 3, but not in the cache of size 4. This is a violation of the inclusion property.

An interesting question is whether it is feasible to use MRCs generated under the assumption of ideal LFU caches (which adhere to the strict inclusion property) to model the miss ratios of practical LFU caches. Figure 2 demonstrates that this is not the case. The figure depicts the MRCs for ideal and practical LFU caches for two workloads. For the MSR `src1` workload [23], the miss ratios deviate substantially for cache sizes between 190GiB and 240GiB. Similarly, for the MSR `web` workload [23], the miss ratios deviate significantly for cache sizes between 38GiB and 46GiB.

Table 1: *Sample trace for LFU caches of sizes* 3 *and* 4 *demonstrating a violation of the inclusion property.*

Time	Access	LFU cache size 3	LFU cache size 4
1	a	a(1)	a(1)
2	b	b(1), a(1)	b(1), a(1)
3	c	c(1), b(1), a(1)	c(1), b(1), a(1)
4	d	d(1), c(1), b(1)	d(1), c(1), b(1), a(1)
5	a	a(1), d(1), c(1)	a(2), d(1), c(1), b(1)
6	d	d(2), a(1), c(1)	d(2), a(2), c(1), b(1)
7	b	d(2), b(1), a(1)	b(2), d(2), a(2), c(1)
8	e	d(2), e(1), b(1)	b(2), d(2), a(2), e(1)
9	f	d(2), e(1), f(1)	b(2), d(2), a(2), f(1)

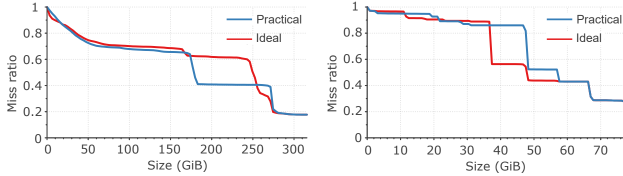


Figure 2: *MRCs for ideal and practical LFU caches for the MSR* `src1` *(left) and* `web` *(right) workloads [23].*

2.3 MRC generation

Mattson's algorithm. Mattson et al. were the first to describe a method capable of constructing a miss ratio curve from an LRU cache's access trace in a single pass [14]. To generate the MRC, Mattson's algorithm maintains an LRU stack of all the accessed objects. Because MRC generation algorithms simply model caches but do not store the values of the accesses, an "object" in this context refers to the referenced key or a hash of the referenced key. Upon each access to an object, if the object has been previously accessed, its *reuse distance* is measured as the number of objects ahead of it in the stack and is recorded in a histogram. The object is then moved to the front of the LRU stack. If an object has not been seen before, the reuse distance is said to be infinity and is recorded as such in the histogram; then a new object is instantiated and inserted at the front of the LRU stack. After the entire trace has been processed, the resulting MRC is generated as the inverse CDF of the histogram. The Kosmo algorithm also uses a histogram of reuse distances to generate MRCs.

Other algorithms have been introduced to improve on Mattson's computational overhead. Olken [15] maintains the objects in a balanced tree, sorted by their last access times to bring the compute complexity from $O(MN)$ to $O(NlogM)$, where M and N are the number of unique and total number of accesses, respectively. Parda [16] extends the Olken algorithm to support the parallel processing of an access trace.

SHARDS. Waldspurger et al. describe an algorithm called *SHARDS* which works in conjunction with an exact MRC generation algorithm, such as Olken. SHARDS uses Olken to generate the MRC, but uses only a sampled subset of the trace [17]. This significantly improves the efficiency with which an MRC can be generated, and while the resulting MRC is approximate, it typically has reasonably low error [17]. Kosmo also uses SHARDS to improve its performance.

SHARDS can be implemented as either fixed-rate or fixed-size. The *fixed-rate* implementation samples the trace using a specific rate, R. The authors of the paper found that an R value of 0.001 results in reasonably accurate MRCs, thus reducing the overhead by a factor of 1,000 [17]. The *fixed-size* implementation extends the fixed-rate implementation by adjusting the sampling rate downward so as to limit the number of objects that exist in the MRC algorithm's internal data structures at any given time to a constant, S_{max}. In our experimentation, we use S_{max} values of 1,024 and 2,048 when running Kosmo (as was done in the SHARDS paper).

The authors of SHARDS noticed that the expected number of sampled accesses, $E[N_S]$, was often not equal to the measured number, N_S. To correct for this, after the access trace has been fully processed, the difference between $E[N_S]$ and N_S is added to the first histogram counter (i.e., the histogram counter for the smallest reuse distance). By applying this correction, the authors achieved significant improvements in the error induced by the sampling algorithm [17].

Miniature Simulations. Waldspurger et al. describe a method of generating MRCs called *Miniature Simulations* (which we will refer to as "MiniSim"), capable of modelling any eviction policy. MiniSim independently simulates caches at varying sizes to obtain the resulting MRC [18].

To generate an MRC using MiniSim, a maximum simulated cache size, C_{max}, is selected. A number of simulated caches (N_C) are then instantiated, each simulating a cache size between C_{max}/N_C and C_{max}. In practice, N_C is often set to 100. Upon each access in a trace, the access is processed by each simulated cache. When the trace is complete, the miss ratios of the simulated caches and each simulated cache's respective size are used to form an MRC.

MiniSim utilizes the sampling method proposed by SHARDS to operate on a small subset of the total trace to improve runtime performance, making it an approximate MRC generation algorithm.

Although MiniSim can generate MRCs for any eviction policy, it has two key shortcomings. First, it has high memory usage as each data point on the curve simulates an instance of a cache, and the different simulations of the caches do not share any of their internal data structures. These caches, especially those with similar sizes, often contain many of the same objects, yet each cache allocates memory for these objects independently. Experimentally, we found that MiniSim used an average of 113MiB to generate a single MRC for the LFU eviction policy, with up to a maximum of 3.1GiB. To reduce this memory usage significantly, one would have to reduce the sampling rate of SHARDS or reduce the number of simulated caches, which in turn would reduce the accuracy of the resulting MRC.

A second key shortcoming is that the range of cache sizes to be simulated must be defined before the input trace is first processed and cannot be modified while the simulation is ongoing. This is limiting when generating MRCs online for live

workloads, where the workload's working set size is unknown ahead of time. Because the maximum simulated cache size C_{max} cannot be modified after the simulation begins, a large, worst case, value is typically selected to ensure the access trace's working set size (i.e., the cache size required to store all unique objects) is likely to be captured. This prevents MiniSim from being able to focus the sizes of the simulated caches to regions of the MRC which lie within the workload's actual working set size.[3] For example, although Twitter tends to overprovision its caches [3], of the publicly available access traces in the Twitter dataset, 25% have a working set size of less than 2GiB. If a large C_{max} value, such as 200GiB, is selected to model one of these access traces, the simulated caches are sized in increments of $200\text{GiB}/100 = 2\text{GiB}$, preventing points of interest on the MRC from being observable. This requires MiniSim to be configured with a large number of simulated caches as reducing this to a smaller value will further reduce the resulting MRC's granularity.

3 Kosmo

We now present Kosmo, an MRC generation algorithm capable of generating approximate MRCs for a variety of eviction policies simultaneously. We begin by presenting the algorithm in general terms (§3.2) and then describe several optimizations that significantly improve its efficiency (§3.3). We then describe the Kosmo algorithm for the LFU eviction policy (§3.4) specifically, followed by the required extensions to support other evictions policies (§3.5). We then show how Kosmo can be extended to support variable object sizes (§3.6) and TTLs (§3.7). Finally, we describe how Kosmo can generate MRCs for multiple eviction policies simultaneously (§3.8).

Both Kosmo and MiniSim simulate caches of different sizes to generate an MRC. The key difference is that MiniSim maintains a stack for each cache throughout the duration of the simulation, while Kosmo reconstructs the stacks dynamically, only as needed. MiniSim keeps track of the miss ratios of the different caches and constructs the MRC using these miss ratios once it has processed the entire access trace, while Kosmo uses an approach similar to Mattson: it records stack distances encountered in a histogram and, in the end, constructs the MRC from the histogram.

Further, MiniSim always simulates the same pre-configured cache sizes, regardless of the working set size of the access trace, while Kosmo simulates a different set of cache sizes on each access, the largest simulated cache size being the reuse distance of the currently accessed object minus one. This allows Kosmo to generate MRCs with similar error rates to that of MiniSim while simulating far fewer caches, which leads to lower memory and compute overheads for Kosmo.

The simulated caches in an instance of MiniSim do not share any internal data structures, therefore an object may exist simultaneously in the stacks of multiple caches, causing MiniSim to consume large amounts of memory. In contrast, Kosmo maintains the data representing an object only once in a global data structure. Each object in this data structure contains the minimal amount of data required to allow the stack of a cache of any size to be reconstructed dynamically.

3.1 Kosmo data structures

Kosmo maintains all objects ever accessed in a data structure called the *global table*, implemented as a dynamic hash table. Each object in the global table has an associated *eviction map* which, in turn, consists of a set of *eviction records*. Whenever an object is evicted from any of the caches (of different sizes) being simulated, an eviction record is added to the eviction map of the object. This eviction record includes or registers the size of the cache from which the object was evicted, as well as other policy-specific information described further below. Using an object's eviction map, Kosmo is able to determine at any time whether the object exists in a cache of a specified size. If it exists in the cache, Kosmo can determine its position within the cache's internal data structure, referred to as the cache's *stack*, using the policy-specific information in the eviction records. An eviction map also holds a reference to the associated object, allowing it to access the object's properties, such as the last access time or ideal frequency count in the case of LFU caches.

Eviction maps are eviction policy-specific and must be implemented on a per-policy basis. However, all eviction maps support three primary operations:

1. For a given object, identify the size of the smallest cache that contains the object.
2. For its associated object, calculate the object's sorting key, given a cache size, S. The sorting key is calculated using policy-specific information in the eviction records, allowing Kosmo to properly order objects in the stack of the cache of size S it is reconstructing.
3. Insert a new eviction record.

The specific implementation of eviction maps for the LFU eviction policy is described in §3.4. The implementations for the FIFO, 2Q, LRFU, LRU, and MRU policies are described in §3.5. The implementations for the LRU and MRU policies are provided to demonstrate Kosmo's generality and are not included in the experimental analysis.

3.2 The Kosmo algorithm

We first describe a variant of the Kosmo algorithm that is highly inefficient. Optimizations that make it efficient are described in §3.3. Upon each access, the Kosmo algorithm performs the following sequence of steps:

1. Calculate the reuse distance D of the accessed object and

[3] The sizes of MiniSim's simulated caches can be configured non-uniformly [18], though this would require knowledge of either the shape of the MRC or a specific point of interest around which to cluster the sizes of the simulated caches (e.g., the current size of the production cache) before processing the access trace. Further, the shapes of some workloads' MRCs can change dramatically over time [20, 32, 33].

update the histogram counter associated with D.

2. Reconstruct the stacks of the caches of sizes $S < D$, for every possible cache size at a byte-level granularity. These are the caches that do not contain the accessed object.
3. Select an object from each reconstructed stack for eviction (to make space for the accessed object) and place a new eviction record in the eviction map of the evicted object.

Using the accessed object's key, its associated eviction map is found using the global table. The object's reuse distance D can be determined using the object's eviction map by finding the smallest sized cache in which, according to the eviction records, the object exists. If the object is not found in the global table, its reuse distance is set to infinity, as it is being accessed for the first time. The histogram counter associated with D is then incremented.

For eviction policies that do not adhere to the inclusion property, an interesting question is: what is the reuse distance of an object? For eviction policies that *do* adhere to the inclusion property, the reuse distance is clear. It is simply the minimal cache size that contains the accessed object; any larger cache will contain the object, while any smaller cache will not. For eviction policies that do *not* adhere to the inclusion property, an object may exist in a cache of size S, but not exist in some caches of size $S' > S$. Nevertheless, we argue that the size of the smallest cache containing the accessed object should be the reuse distance. There are several motivations for this choice. First, this choice allows for an important optimization to ensure eviction maps do not contain a large number of eviction records, which we describe in §3.3. Second, the choice simplifies the calculation of an object's reuse distance. Third, in our experiments, we found it is rare for an access to cause a violation of the inclusion property and maintain this violation for large ranges of cache sizes, which we show in §4.5.

Immediately after an object O is accessed, it must exist at all cache sizes. Hence, for each cache of size S in which the object does not exist when it is accessed, some object needs to be evicted to make space for O. To select an object for eviction, Kosmo reconstructs the stack of the cache by iterating through all objects in the global table. Using each object's eviction map, Kosmo determines if the object exists in the cache (of size S) and, if so, where in the cache's stack the object resides relative to other objects using the objects' sorting keys. Through this process, Kosmo reconstructs the cache's full stack. The object at the top of the reconstructed stack is selected as the object to be evicted and a new eviction record is accordingly inserted in the object's eviction map.

It is clear that Kosmo, as described, is highly inefficient. First, because the global table contains an entry for every object ever accessed, it can grow quite large. Second, to determine which objects need to be evicted from which reconstructed stacks, Kosmo must reconstruct the stack for every cache of size less than the accessed object's reuse distance. Finally, as an eviction record is inserted into an object's eviction map each time it is selected for eviction, the eviction maps may contain a large number of eviction records.

3.3 Optimizations

We describe four optimizations: cache size granularity, eviction record pruning, the use of SHARDS, and parallel stack reconstruction.

Granularity. To reduce the number of cache stacks that need to be reconstructed on each access, a granularity parameter G is introduced. This parameter limits the number of caches that need to be reconstructed on each access to a fixed number. For example, if an object O is accessed and all cache sizes less than or equal to 3GiB are found to not contain O and must therefore be reconstructed to perform the necessary evictions, with a granularity parameter of 100, 100 simulated caches in size increments of $3\text{GiB}/100 = 30\text{MiB}$ are examined.

We experimentally evaluated appropriate values of G. A higher value of G typically means a lower MAE (mean absolute error); however, this also leads to increased computational overhead. Figure 3 shows the experimental results of varying values of G and the corresponding MAEs for all workloads in the MSR dataset [23]. An interesting observation is that Kosmo can achieve a low mean MAE with even a small value of G. As evident in this figure, selecting a G value greater than 10 does not significantly reduce the mean MAE. We therefore conservatively select 10 as our value of G and use it throughout all our simulations.

Upon accessing an object with reuse distance D, Kosmo only simulates G caches of sizes $S < D$. As a result, it is able to achieve a comparable MAE while simulating significantly fewer caches than MiniSim. The accessed object is assumed to already exist in caches of sizes $S \geq D$, so they do not need to be simulated. In contrast, MiniSim simulates all (typically 100) considered cache sizes on each access, regardless of the accessed object's reuse distance.

Eviction record pruning. To reduce the number of eviction records in an object's eviction map, each time a new eviction record is added, indicating the object is being evicted from the cache of size S, all eviction records with cache size $S' < S$ are removed. In doing so, Kosmo is effectively assuming the inclusion property where an object being evicted from a cache of size S will thereafter also not exist at any cache of size $S' < S$. This may introduce inaccuracies for eviction policies which do not adhere to the inclusion property, however, we have found these inaccuracies to be negligible (§4.4).

Pruning drastically reduces the size of each object's eviction map. Figure 4 shows the effect of pruning on the average number of eviction records in objects' eviction maps throughout a typical access trace of a workload. On average, pruning reduces the average size of the eviction maps (i.e., the numbers of records it contains) by a factor of roughly 387. This drastically reduces the memory required to store the eviction records as well as the computational overhead of searching

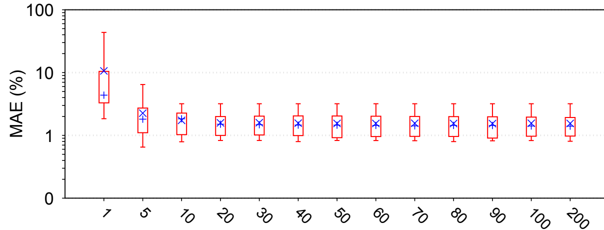


Figure 3: MAEs of varying granularities for all workloads in the MSR dataset [23]. A fixed-sized implementation of SHARDS was used with $S_{max} = 2{,}048$ for granularities between 1 and 200. The top line identifies the maximum result while the bottom line identifies the minimum. The top of the box is the 75th percentile result and the bottom of the box is the 25th percentile result. The $\times$ and $+$ symbols indicate the mean and median MAEs, respectively, for each granularity value.

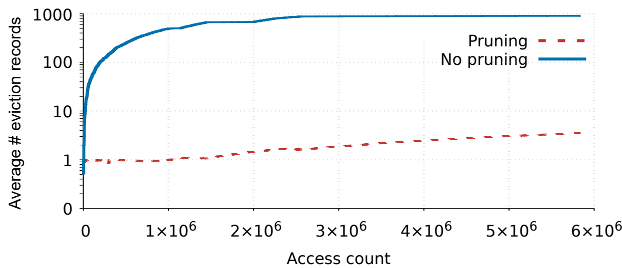


Figure 4: The average number of eviction records per object in the global table throughout the MSR web workload [23] using fixed-rate SHARDS ($R = 0.001$) shown with (dashed red line) and without (solid blue line) pruning. The unsampled access count is shown here (i.e., not the number of sampled accesses).

through the eviction records when determining if the object exists in a cache of size S.

SHARDS. On each access, Kosmo must iterate through all objects in its global table to reconstruct the stacks of various cache sizes. The global table could include billions of objects. For this reason, we use SHARDS to spatially sample the accesses. This lessens the number of objects in the global table and reduces the stack reconstruction time. The fixed-size variant of SHARDS (§2.3) is particularly useful for Kosmo as it limits the size of the global table to a known constant: the S_{max} value of SHARDS.

Parallel stack reconstruction. As the reconstruction of a cache's stack does not modify the global table, the reconstruction of the stacks of multiple caches can all be done in parallel. This improves the response time of Kosmo, but increases the memory overhead because the stacks of all caches exist in memory simultaneously. This creates a trade-off between throughput and memory usage. However, as we show in §4.4, Kosmo uses significantly less memory than MiniSim, even when reconstructing stacks in parallel.

3.4 Kosmo for LFU

To support the LFU eviction policy specifically, Kosmo maintains the following information in its data structures. First,

Algorithm 1: Eviction map for LFU cache object.

```
   Ref      : object
   Record   : map → Map<cache_size, count>
1  Eviction Map LFU:
2      Function Insert(cache_size):
3          map.insert(cache_size, object.global_count)
4      Function FindSmallestExisting():
5          for record in map do
6              if record.count == object.global_count then
7                  return record.size + 1
8          return object.size
9      Function GetSortingKey(cache_size):
10         rec ← map.find(≥ cache_size)
11         if !rec then
12             return object.global_count
13         return object.global_count − rec.count
```

each object in the global table maintains a timestamp of when the object was last accessed and a counter referred to as the object's *global count*. This counter is incremented by one each time the object is accessed and is therefore the same as the object's frequency count in an ideal LFU cache. Second, each eviction record in the object's associated eviction map contains the size of the cache from which the object was evicted, and the object's global count value when the eviction occurred. The latter makes it possible to infer the object's frequency count for a specific cache size, referred to as the *local count*, as it would be in a practical implementation of a cache of that size. Algorithm 1 contains the pseudocode for the eviction map's three main operations.[4]

We found that while a practical LFU cache regularly violates the strict inclusion property, it does not violate the (non-strict) inclusion property often. Although objects in the stacks of caches of different sizes may be in different orders, typically, the objects in each cache's stack are a subset of the objects in the stack of a larger cache. Experimentally, we noticed that, on average, only 1.49% of accesses to an LFU cache cause the (non-strict) inclusion property to be violated. With Kosmo, we therefore assume that the (non-strict) inclusion property holds for practical LFU caches (unlike Mattson which assumes the strict inclusion property holds) and show in our experimentation results that this produces negligible errors in the resulting MRCs.

To obtain the reuse distance of an object O when it is accessed, we search for O's eviction records to identify the record with the largest cache size S wherein the record's count value is equal to the object's current global count. This record indicates the object has not been accessed in a cache of size S since O was last evicted (at which time this record was inserted) and therefore the object does not exist in any caches

[4] Our descriptions here assume fixed-sized objects, though the algorithms shown in the listings accommodate variable-sized objects as described in §3.6 (i.e., `object.size` [variable-sized] in the listings corresponds to `1` [fixed-sized] in the text).

of sizes $S' \leq S$. The object's reuse distance is then $S+1$. If no eviction record which satisfies this condition is found, we can conclude the object exists at all cache sizes, so the reuse distance is 1.

To reconstruct the stack of a cache of size S, we use the eviction map for each object in the global table to (i) determine if the object exists in a cache of this size, and (ii) if the object exists, calculate its sorting key. To determine if the object exists in a cache of size S, we simply check if S is greater than or equal to the object's reuse distance. If the object is found to exist, its sorting key is the object's local count paired with its last access time. Once calculated, the object then can be placed in the correct position in the reconstructed stack.

To calculate the local count of an object O in a cache of size S, we search O's eviction map for an eviction record with the smallest cache size S' that satisfies $S' \geq S$. If no such record exists, then O has never been evicted from any cache of size $S' \geq S$ since O was first accessed, therefore its local count is equal to its global count. Otherwise, if an eviction record with size $S' \geq S$ is found, the local count is O's global count minus O's global count when it was evicted from the cache of size S' (i.e., the count value in the eviction record), given that the local count should equal the number of accesses to O since its last eviction.

After reconstructing the LFU stack of a cache, Kosmo selects the object at the top of the stack (i.e., the object with the smallest sorting key) for eviction. A new eviction record is inserted with the cache size and the object's global count into said object's eviction map.

3.5 Other eviction policies

The Kosmo algorithm, as described in §3.2, is not designed for any specific eviction policy. Kosmo can generate MRCs for other eviction policies by using the same process, but using policy-specific implementations for the eviction maps. Here, we describe eviction maps for five other policies, namely FIFO, 2Q, LRFU, LRU, and MRU as examples.

FIFO. The design of the FIFO eviction map is fundamentally different than that of the LFU eviction map. A FIFO eviction record indicates the cache sizes for which an object does exist in the cache, whereas an LFU eviction record indicates the cache sizes for which it does not. Each eviction record in a FIFO eviction map records the cache size S and the *entry time* of the object at size S. The record indicates an object's entry time for caches of sizes S', where $S \leq S' < S_{next}$ and S_{next} is the cache size stored in the eviction record with the next largest cache size. Algorithm 2 contains the pseudocode for the eviction map's three main operations.

Algorithm 2: Eviction map for FIFO cache object.

```
   Ref    : object
   Record : map → Map < cache_size, timestamp >
1  Eviction Map FIFO:
2      Function Insert(cache_size):
3          rec ← map.find_largest(≤ cache_size)
4          rec.cache_size = cache_size + 1
5          map.remove(≤ cache_size)
6      Function FindSmallestExisting():
7          rec ← map.find_smallest()
8          return rec.cache_size
9      Function GetSortingKey(cache_size):
10         rec ← map.find_largest(≤ cache_size)
11         if !rec then
12             return 0
13         return rec.timestamp
```

On every access to an object, as the object must exist at all cache sizes immediately after it has been accessed, a new eviction record with a cache of size 1 is inserted into the object's associated eviction map if a record with a cache size of 1 does not already exist. This eviction record stores the entry time (i.e., the current time) of the object for any cache size smaller than the object's reuse distance (i.e., any cache size at which the object does not currently exist).

An eviction record also gets added for an object O when it gets evicted. To insert a new eviction record for a cache of size S into the eviction map of O, we first locate the record with the largest cache size S' such that $S' \leq S$. This record holds the object's entry time E into a cache of size S. We then insert a new eviction record with cache size $S+1$ and entry time E into the eviction map to indicate this object is being evicted from all caches of sizes $S' \leq S$. Finally, we perform pruning by removing all records with cache sizes $S' \leq S$.

The reuse distance of an accessed object can be determined using the object's eviction map as the smallest cache size S contained in the eviction records. By definition, a cache of size S is the smallest cache which contains the object.

When reconstructing the stack of a cache of size S, the objects' sorting keys are their entry times into the cache. To determine the entry time of an object O in a cache of size S, we find the eviction record with the largest cache size S' in O's eviction map such that $S' \leq S$. The entry time contained in this record is the object's entry time for a cache of size S.

2Q. The 2Q eviction map maintains two sets of eviction records: one corresponding to the A1 stack, and the other corresponding to the Am stack. While 2Q further partitions the A1 stack into two stacks, A1in and A1out, we model both as a single combined FIFO stack with a single set of eviction records since objects evicted from A1in are placed at the head of A1out. The position in the combined FIFO stack determines whether an object being accessed a second time should be promoted to Am. Each record in the A1 set of eviction records holds the same information as FIFO eviction records and can be used to determine the entry time of an object in the A1 stack. For an object to exist in the Am stack, it must have been accessed at least twice. We can track the number of accesses of each object at varying cache sizes using the same method we used for LFU eviction records. The handling

Algorithm 3: Eviction map for 2Q cache object.

```
  Ref     : object
  Record  : a1_map → Map < cache_size, timestamp >
  Record  : am_map → Map < cache_size, count >
  Record  : Kin, Kout
1 Eviction Map 2Q:
2   Function Insert(cache_size):
3     insert_a1(cache_size * (Kin + Kout))
4     insert_am(cache_size)
5   Function FindSmallestExisting():
6     a1_rec ← a1_map.find_smallest()/(Kin + Kout)
7     am_rec ← am_map.find_smallest(count ≥ 2)
8     if !am_rec then
9       return a1_rec.cache_size
10    return am_rec.cache_size
11  Function GetSortingKey(cache_size):
12    a1in_size ← cache_size * Kin
13    a1out_size ← cache_size * (Kin * Kout)
14    a1in_rec ← a1_map.find_largest(≤ a1in_size)
15    a1out_rec ← a1_map.find_largest(≤ a1out_size)
16    if !a1out_rec then
17      return A1(a1in_rec.timestamp)
18    am_exists ← map.find_any(> a1in_size and
        ≤ a1out_size and object.global_count − count ≥ 2)
19    if !am_exists then
20      return A1(a1out_rec.timestamp)
21    return Am(object.last_access_time)
```

of an access to the eviction map's associated object and the insertion of a new eviction record are done using the same methods previously described for the LFU and FIFO eviction maps. Algorithm 3 contains the pseudocode for the 2Q eviction map's three main operations.

An object will have different a reuse distance depending on whether it is in the A1 or the Am stack. Therefore we calculate two different reuse distances assuming the object is in each stack and select the smaller value (as the cache associated with the larger reuse distance will inherently also contain the object according to the inclusion property). Similar to the FIFO eviction map, we find the smallest A1 stack of size S_{A1} which contains the object by finding the eviction record in the A1 eviction record set with the smallest cache size. The corresponding cache size which contains the object is then $S_{A1}/(Kin+Kout)$. We then search the Am eviction record set for the eviction record with the smallest cache size which has a local count ≥ 2. This eviction record corresponds to the smallest cache of size S_{Am} which contains the object in the Am stack. If no such record exists, the object must exist in the A1 stack and the previously calculated cache size corresponding to the A1 stack is selected as the reuse distance. Otherwise, the reuse distance is $min(S_{A1}/(Kin+Kout), S_{Am})$.

When reconstructing a 2Q stack, we reconstruct the A1 and Am stacks separately. Unlike the previously described policies, an object in a 2Q cache can exist in one of three different stacks: A1in, A1out, or Am. To determine in which stack an object exists for a cache of size S, we use the object's 2Q eviction map to determine its FIFO entry time if it were to exist in the A1in or A1out stack and its local count if it were to exist in the Am stack. As the size of the A1in and A1out stacks are only a ratio of the total cache size, we find the object's entry time in the A1in and A1out stacks for caches of size $S_{A1in} = S * Kin$ and $S_{A1out} = S * (Kin + Kout)$, respectively.[5] If no entry time is found for the object in the A1out stack, it must exist in the A1 stack with the associated A1in entry time. If an entry time is found for the object in the A1out stack, we search the Am eviction records to determine if the object has a local count ≥ 2 for a cache of size $S_{A1in} < S' \leq S_{A1out}$. If such a record exists, it indicates the object has been accessed at least twice while existing in the A1out stack and is therefore in the Am stack. Otherwise, it is in the A1 stack with the associated A1out entry time.

The implementation of an eviction map for the S3-FIFO eviction policy [34] is a simple adaptation of the eviction map for the 2Q eviction policy.[6] We believe a similar technique would work for other multi-stack eviction policies (e.g., ARC [35]) and leave this for future work.

LRFU. The LRFU eviction map is implemented similarly to that of FIFO. Each object in a cache has an associated CRF value (§2.1). Each eviction record in an LRFU eviction map contains the cache size S and the CRF value of the object at S. Such a record identifies the CRF value of the associated object for caches of sizes S', where $S \leq S' < S_{next}$ and S_{next} is the cache size stored in the eviction record with the next largest cache size.

Similar to when using a FIFO eviction map, the reuse distance of an accessed object can be determined using the object's eviction map as the cache size contained in the eviction record with the smallest cache size.

On each access to object O, a new eviction record is added to O's eviction map with S set to 1 and CRF set to $F(0)$. Moreover, the CRF value of each eviction record is updated using the current CRF value and the object's last access time. Each time an object O is evicted, a new eviction record is inserted into O's eviction map. This process is identical to that of a FIFO eviction map.

When reconstructing the stack of a cache of size S, the

[5] Here, we use the combined size of the A1in and A1out stacks when searching for the object's entry time in the A1out stack as the stacks behave as one coherent FIFO stack.

[6] We have also designed eviction maps to support the S3-FIFO eviction policy [34]. It uses three sets of eviction records: one for the "small" stack, one for the "main" stack, and one to track the number of accesses to each object (as with LFU and 2Q eviction records). When reconstructing the S3-FIFO stack, Kosmo reconstructs a separate stack for the "small" and "main" stacks (the "ghost" stack is inherently maintained by the object's existence in Kosmo's global table). When updating the eviction maps of objects, if an object in the "main" stack is selected for eviction though has a local count ≥ 1, it's local count is reduced by 1 and its eviction record is not further updated. We measured similar performance results (including throughput, memory usage, and accuracy) as with that of 2Q in §4.4. Further details on the implementation of Kosmo's eviction map for the S3-FIFO eviction policy and its evaluation are omitted due to space limitations.

objects in the stack are ordered by their CRF values from smallest to largest. To determine the CRF value of an object O in a cache of size S, we find the eviction record with the largest cache size S' in O's eviction map such that $S' \leq S$. The CRF value contained in this record is the object's CRF value at a cache of size S.

LRU. Because the LRU eviction policy adheres to the strict inclusion property, the order of the objects in the simulated caches (of different sizes) will always be the same and can be determined using the objects' last accessed times.[7] By simply storing the reuse distance D of each object as the sole eviction record in its eviction map, Kosmo can reconstruct the stack of a cache of size S by first determining which objects exist in the cache (any object where its reuse distance $D' \leq S$), then ordering the objects that exist by their last access times.

As an object is moved to the front of the LRU stack each time the object is accessed, immediately after, it will exist in the stacks of all caches, regardless of size, until it is again evicted from a cache. Therefore, each time an object is accessed, its associated eviction map updates the object's stored reuse distance to 1 to indicate it now exists at all cache sizes.

Upon each access to an object, to select which objects to evict from the reconstructed cache stacks, an object may be selected for eviction from multiple of these caches. As the object's eviction map has only one eviction record, the object is evicted from the cache with the largest size. In doing so, as LRU adheres to the inclusion property, Kosmo is effectively evicting the object from all caches of smaller sizes as well.

MRU. The implementation of an eviction map for the MRU eviction policy is virtually identical to that of the LRU policy except that the sorting key in the MRU eviction map is the negative value of the object's last access time.

3.6 Variable object sizes

The Kosmo algorithm described thus far generates MRCs assuming the cache is being used for fixed-size objects. However, modern applications use caches to store objects of varying size (e.g., key-value caches). As such, the MRCs generated by these algorithms may not adequately represent the miss ratios experienced by the caches under these workloads. Figure 5 demonstrates the difference in MRCs for the same workload when taking variable-sized objects into account versus not taking them into account. It is evident that these two MRCs differ significantly and variable-sized objects should be accounted for accordingly in MRC generation algorithms.

With fixed-sized objects, Kosmo inserts an eviction record into the eviction map of only one object when a new object is accessed. However, with variable-sized objects, more than one object may need to be evicted. A simple modification

[7] In practice, one would always generate an MRC for the LRU eviction policy using SHARDS and Olken as it is far more efficient than any other known methods. We present a method of generating this MRC using Kosmo simply to demonstrate Kosmo's generality.

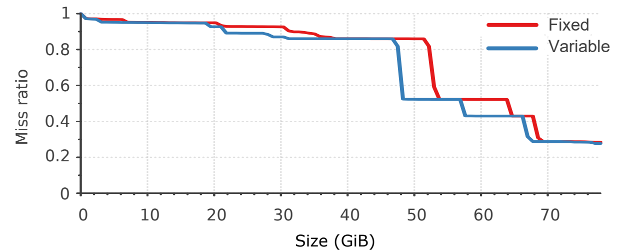


Figure 5: MRCs for the MSR web workload [23] for fixed versus variable-sized objects using the LFU eviction policy.

allows the algorithm to handle variable-sized objects. While a cache's stack is being reconstructed, the cache's used size is calculated by summing the size of all objects which exist in the cache. Objects are then evicted from the top of the stack until the total used size of the cache is less than or equal to the cache's size.

3.7 TTLs

Taking *time-to-live* (TTL) parameters into account can significantly affect the resulting MRC [36]. Minor modifications to the Kosmo algorithm can allow for the support of TTL parameters for objects. When an object is first accessed, a corresponding *expiry time* is calculated based on its TTL. If the TTL is 0, no expiry time is specified. Expiry time describes the time at which the object should be evicted from all caches, regardless of size. To handle this in Kosmo, when iterating through the global table upon each access to reconstruct a cache's stack, each object's expiry time is compared against the current time (i.e., the time of the current access). If the object has not expired and exists in the cache, it is added to the stack; otherwise, it is excluded.

3.8 Simultaneous MRC generation

One key advantage of Kosmo is its ability to generate MRCs for multiple eviction policies simultaneously in a single pass. In the previous descriptions of eviction maps, each object in the global table has one associated eviction map. The type of eviction map used is based on the eviction policy for which an MRC is being generated. A simple extension to the global table to allow each object to have multiple associated eviction maps – one for each eviction policy – allows Kosmo to reconstruct the internal stack of any cache size with any eviction policy for which an eviction map has been defined.

Minor modifications to the previously described Kosmo algorithm (§3.2) must be made to support multi-policy MRC generation. As an MRC must be generated for each eviction policy, the algorithm must maintain a separate histogram per policy. Upon access, an object's reuse distance is calculated for each eviction policy and its corresponding histogram is updated. Moreover, each policy being considered will need its own cache simulations, and each cache's stack is reconstructed in the policy-specific way independently.

4 Evaluation

We evaluated Kosmo using 52 publicly-accessible cache access traces from MSR [23], Twitter [3], and SEC [37, 38]. Table 2 shows a summary of the datasets we used in our evaluation. For the Twitter dataset, we used the recommended traces as specified by Twitter [39] as well as 7 other randomly selected traces in the dataset.[8] Similar to prior studies, we only considered the GET/READ accesses in each trace [20, 40].

For LFU and FIFO, we evaluated Kosmo's performance using all 52 access traces. For 2Q and LRFU, we used all access traces in the MSR and SEC datasets, and 5 randomly selected access traces the Twitter dataset.[9]

We ran both Kosmo and MiniSim with three configurations of SHARDS: one fixed-rate and two fixed-size. The authors of SHARDS noted that for fixed-rate SHARDS, an R value of 0.001, and for fixed-size SHARDS, an S_{max} value of 2,048 produce reasonably accurate MRCs [17]. We selected these same values for our simulations, but also used fixed-size SHARDS with an S_{max} value of 1,024 to examine the memory and throughput benefits as well as the reduction in accuracy. For all fixed-size configurations of SHARDS, we used an initial sampling rate of $R = 0.1$ as we found this produces accurate results.

4.1 MiniSim implementation

We implemented the MiniSim algorithm as described by the original authors of the paper, with the same configuration parameters [18]. The authors only describe MiniSim configured using the fixed-rate SHARDS variant; therefore, we extended MiniSim to also support the fixed-sized SHARDS variant.

Unlike fixed-rate SHARDS, which keeps the sampling rate R constant throughout the access trace, fixed-size SHARDS gradually decreases R to ensure that at any given time there are at most S_{max} distinct objects in the MRC generation algorithm's internal data structures. SHARDS tracks these unique objects in a set S. To extend MiniSim to support fixed-size SHARDS, we initially scale each simulated cache size by the initial sampling rate R. For each access, if the sampling rate R is decreased to R_{new}, we remove all objects no longer in S from all simulated caches. We then rescale the size of each simulated cache by R_{new} using the eviction policy of the cache. A key insight is that when rescaling the size of a simulated cache, we also rescale the cache's access counter (i.e., the number of accesses the cache has observed) and hit counter by the factor R_{new}/R_{old}.

The implementation of MiniSim described in the original paper statically allocates the required memory for each simulated cache before processing an access trace. This is possible as the sampling rate R, and thus the size of each simulated cache, is fixed. In our extension, to support a varying sampling rate, we allocate memory dynamically so as to be able to release memory when R decreases.

Table 2: Access trace datasets used in our simulations.

Dataset	Access traces	Total accesses
MSR [23]	13	434,212,008
Twitter [3]	24	99,200,180,813
SEC [37, 38]	15	26,482,889,754

Our LFU implementation follows a well-known algorithm optimized for throughput to allow for constant time complexity for each access [41]. Our implementation of 2Q follows that described by the original authors [29]. We used Kin and Kout values of 25% and 50%, respectively, for both our Kosmo and MiniSim simulations. These were the same values used by the original authors. Our implementation of the LRFU eviction policy follows the description in the original paper [27]. We arbitrarily selected a λ value of 0.5 for our experiments though experimented with other values of λ, such as 0.001, and found the results to be similar.

4.2 Environment

All experiments were done on Ubuntu 22.04.2 with an AMD Ryzen Threadripper 3990x (64 cores) with 256GB of $DDR4-3200MHZ$ DRAM. The access traces were stored in binary format on a Sabrent Rocket Q 8TB. Both Kosmo and MiniSim use a thread pool with separate threads for each of Kosmo's reconstructed stacks and MiniSim's simulated caches. We tested various thread pool sizes and noticed the best performance for MiniSim when the thread pool's size was equal to the number of cores. Kosmo's performance remained the same after the thread pool's size exceeded the configured granularity.

4.3 Metrics

Three metrics were used in the evaluation of the algorithm: memory usage, throughput, and accuracy.

Memory usage. To measure the memory usage of each algorithm, for each access trace, we ran the algorithm in an isolated process and measured the high water mark [42] after it had processed the entire access trace. This metric has been used in prior work to evaluate the memory usage of MRC generation algorithms [17].

Throughput. To measure the throughput of each algorithm, for each access trace, we divided the total runtime by the number of accesses in the trace. IO time is excluded from the measurement of the total runtime.

Accuracy. To measure the error of both Kosmo and MiniSim, we calculated the mean absolute error (MAE) of each of the generated MRCs using the corresponding exact MRC. As no algorithm exists capable of generating exact MRCs for the LFU, FIFO, 2Q, and LRFU eviction policies, we performed 100 full simulations of caches of varying size (evenly dis-

[8] The randomly selected traces are: `cluster1`, `cluster3`, `cluster8`, `cluster10`, `cluster26`, `cluster50`, and `cluster53`.

[9] The randomly selected traces are: `cluster7`, `cluster22`, `cluster31`, `cluster45`, and `cluster50`.

tributed over the access trace's working set size) for each policy and for each access trace. These 100 points are the same points selected when running MiniSim. To measure the error, we found the MAE by calculating the difference between the exact MRC and the approximate MRCs generated by Kosmo and MiniSim at these points.

4.4 Results

Figures 6-8 show the performance results of Kosmo and MiniSim. For each algorithm, the range of results for the various traces in the datasets is shown.

Figure 6 shows the memory usage results of Kosmo and MiniSim for the LFU, FIFO, 2Q, and LRFU eviction policies. We found that Kosmo uses an average of 3.6 times less memory, and up to 36 times less in the extreme case. Figure 7 shows the throughput results of Kosmo and MiniSim for the LFU, FIFO, 2Q, and LRFU eviction policies. We found that Kosmo has an average throughput 1.3 times higher than that of MiniSim. Notably, for the 2Q eviction policy, Kosmo has a lower average throughput than MiniSim (0.54 times that of MiniSim). This is attributed to Kosmo reconstructing two stacks (A1 and Am) on each access.

Figure 8 shows the MAE results of Kosmo and MiniSim for the LFU, FIFO, 2Q, and LRFU eviction policies. We found Kosmo and MiniSim to typically generate MRCs with similar accuracy. Across all simulations, Kosmo and MiniSim had an average MAE within 0.25% of one another. Although Kosmo generates MRCs with lower MAEs, on average, for LFU and LRFU (0.16% and 0.86% lower for LFU and LRFU, respectively), it generates MRCs with higher MAEs, on average, for FIFO and 2Q (0.44% and 1.56% higher for FIFO and 2Q, respectively). This is attributed to the higher rates of violations of the inclusion property for FIFO and 2Q, which we show in §4.5. Further, although the average MAE for the MRCs generated by Kosmo for 2Q is 1.56% higher than those generated by MiniSim, the median is only 0.35% higher. This is attributed to the high MAE of one access trace, `src1` in the MSR dataset [23], which has an unusually high MAE. This access trace violates the inclusion property at a significantly higher rate than other access traces.

To evaluate the CPU usage of Kosmo and MiniSim, we measured the CPU time per access for each access trace. Figure 9 shows that Kosmo's CPU time per access is roughly 1.85 times higher than that of MiniSim for LFU and 2 times higher for FIFO, 2Q, and LRFU. The inconsistency between the lower average CPU time per access of MiniSim than that of Kosmo, and the higher average throughput of Kosmo than that of MiniSim can be attributed to MiniSim's threads idling more frequently than Kosmo's threads.

To evaluate the effects of varying the number of MiniSim's simulated caches on its performance, we also tested MiniSim with 20 and 50 simulated caches for the LFU eviction policy. With 20 simulated caches, MiniSim consumes roughly 1.2 times the memory of Kosmo on average and exhibits roughly similar throughput, however it has roughly 2 times the MAE of Kosmo. With 50 simulated caches, MiniSim consumes roughly 2.3 times the memory of Kosmo and has 10% lower throughput with roughly identical MAE. Notably, as discussed in §2.3, the C_{max} value of MiniSim must be selected before

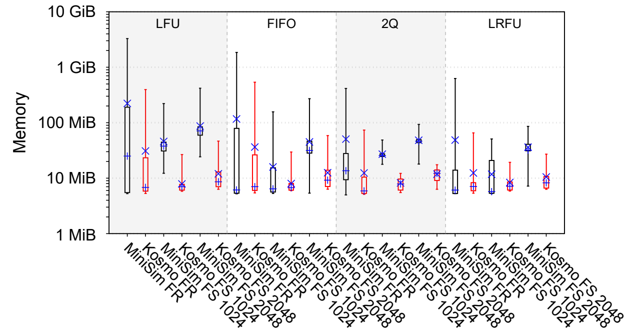


Figure 6: Memory usage of Kosmo and MiniSim for all eviction policies. Note the logarithmic scale of the y-axis.

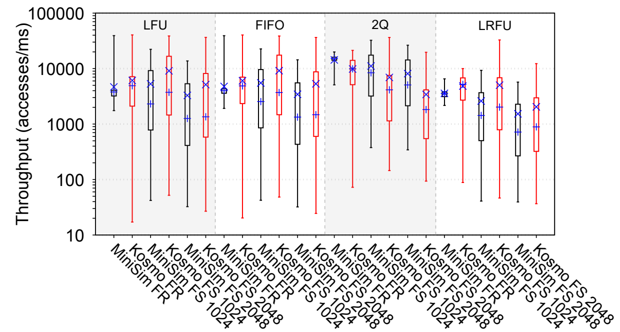


Figure 7: Throughput of Kosmo and MiniSim for all eviction policies. Note the logarithmic scale of the y-axis.

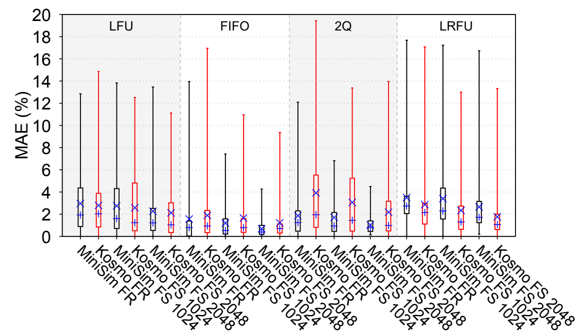


Figure 8: MAE of Kosmo and MiniSim for all eviction policies.

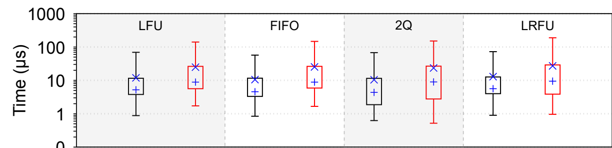


Figure 9: CPU time per access for the LFU, FIFO, 2Q, and LRFU eviction policies for MiniSim (left, black) and Kosmo (right, red) using fixed-sized SHARDS (S_{max} = 2,048).

knowledge of the access trace, therefore using a low number of simulated caches such as these may result in unobservable points of interest on the MRC.

4.5 Inclusion property violations

Figure 10 shows the percentage of accesses for which a violation of the inclusion property occurs (i.e., accesses which reference an object that does not exist in a cache of size S though exists in a cache of size $S' < S$) for the LFU, FIFO, 2Q, LRFU, and MRU eviction policies across all access traces in the MSR dataset [23]. We found these violations by simulating 100 caches of varying sizes evenly distributed over the access trace's working set size and, for each access, finding the smallest simulated cache in which the object exists, then searching for a larger cache in which it does not.

To examine the severity of these violations, indicated by the difference between the size of the smaller cache wherein an object exists and the larger cache wherein it does not, we repeated these simulations with 50 and 10 simulated caches. This increases the size intervals between the simulated caches and thus, if the number of violations remains high, we can infer the violations occur in large ranges of cache sizes.

For the LFU eviction policy, we found that 1.49% of accesses violated the inclusion property when measured with 100 simulated caches. This reduces by 40.91% and 99.3% to 0.88% and 0.01% when measured with 50 and 10 points, respectively. For the FIFO eviction policy, we found that 20.61% of accesses violated the inclusion property with 100 points, reducing by 18.46% and 77.23% to 16.81% and 4.69% for 50 and 10 points, respectively. For the 2Q eviction policy, we found that 10.49% of accesses violated the inclusion property with 100 points. This reduces by 39.39% and 84.25% to 6.36% and 1.65% for 50 and 10 points, respectively. We found that violations of the inclusion property are rare for the LRFU eviction policy (0.08% of accesses violated the inclusion property with 100 points). The higher rates of violations of FIFO and 2Q can explain the higher MAEs of MRCs generated by Kosmo for these policies, however these errors are typically negligible. Interestingly, we found that MRU violates the inclusion property at a higher rate than the other evaluated eviction policies with an average of 29.12% when simulated with 100 points, while MRU is often considered to not violate the inclusion property [18, 43–45].[10]

5 Related work

Much prior work has focused on improving the performance of in-memory caches [29, 32, 33, 46–55]. Many studies have suggested new eviction policies to improve on observed limitations of policies such as LRU [27, 29, 35, 50, 53, 56–60].

There have been many proposed MRC generation algorithms [14–20, 30, 40, 44, 61–64], however, these are largely

[10]We note that violations of the inclusion property only occur for the MRU eviction policy when considering variable-sized objects.

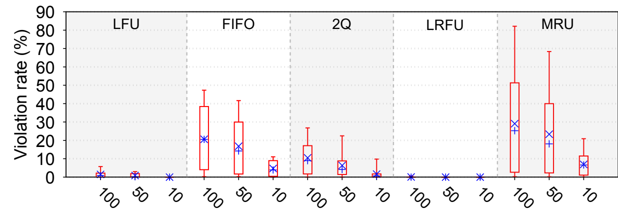


Figure 10: Ratio of accesses for which a violation of the inclusion property occurs for the LFU, FIFO, 2Q, LRFU, and MRU eviction policies across all workloads in the MSR dataset [23] when simulated with 100, 50, *and* 10 *points.*

focused on the LRU eviction policy. Beckmann and Sanchez describe a probabilistic method of generating MRCs for other age-based eviction policies [61], such as *protecting distance based policy* (PDP) [53] or *inter-reference gap distribution replacement* (IGDR) [59].

Yu et al. propose an extension to MiniSim, called *DF-Shards*, to modify the number of simulated caches during runtime [44]. We found that the cost of instantiating a new simulated cache significantly reduces the throughput of DF-Shards making it unsuitable for online MRC generation.

MRCs are widely used to improve the performance of caching systems [46, 65–69]. Talus partitions caches to remove identified cliffs in MRCs [46]. Cliffhanger identifies and flattens cliffs noticed in an MRC in real-time, while processing an access trace [32]. mPart uses MRCs to manage the allocation of caches in multi-tenant caching servers [66]. Dynacache also uses MRCs to manage cache allocation; however, the authors also note that modifying the eviction policy in real-time can improve cache performance [68].

6 Concluding remarks

In this paper, we propose Kosmo, a novel method for the simultaneous generation of miss ratio curves (MRCs) for multiple eviction policies. We showed that the current method of generating MRCs for eviction policies that do not adhere to the strict inclusion property have significant memory overhead and are therefore not suitable for online MRC generation. Our experimental results show that Kosmo uses significantly less memory than MiniSim configured with 100 simulated caches while maintaining similar accuracy. Kosmo uses 3.6 times less memory than MiniSim on average, up to 36 times less in the most extreme case. Kosmo has an average throughput 1.3 times that of MiniSim.

In the future, we plan to expand Kosmo's supported eviction policies to more complex policies, such as LHD [50], LIRS [70], or ARC [35]. We also plan to improve on Kosmo's throughput by reducing its computational overhead through the use of more specialized data structures.

Acknowledgements. We thank the reviewers for their constructive comments. We particularly thank our shepherd Carl Waldspurger, whose guidance was instrumental in significantly improving the paper.

A Artifact Appendix

Abstract

The Kosmo artifact provides our implementations of both Kosmo and MiniSim which were used to generate the results presented in this paper. The artifact repository includes three tools: one to calculate an access trace's working set size, one to compute an access trace's accurate MRC for a given eviction policy, and one to generate an approximate MRC (while measuring memory usage, throughput, and accuracy) for both Kosmo and MiniSim. Specific details on each tool's usage can be found in the artifact's README.

Scope

In our evaluation we make the following claims which can be verified by this artifact:

- Kosmo has lower memory overhead than MiniSim by a factor of 3.6 on average, up to a factor of 36.
- Kosmo has a higher throughput than MiniSim by a factor of 1.3 on average.
- Kosmo has a roughly equivalent MAE to MiniSim.

Contents

The artifact compiles to three binaries (further details, including the specific usage of each tool and format of input data is provided in the artifact's README):

1. **wss**: This tool calculates the working set size of a given access trace.
2. **accurate**: This tool runs full simulations to compute the accurate MRC for a given access trace.
3. **mrc**: This tool runs Kosmo or MiniSim (or both) to generate an MRC for a given access trace.

Hosting

The artifact can be found at: 10.5281/zenodo.10569925.

Requirements

The artifact was compiled and tested using `Rust v1.77.0-nightly` and depends on `Gnuplot v5.4` to generate plots.

References

[1] J. Mertz and I. Nunes, "Understanding Application-Level Caching in Web Applications: A Comprehensive Introduction and Survey of State-of-the-Art Approaches," *ACM Computing Surveys*, vol. 50, no. 6, pp. 1–34, Nov. 2017.

[2] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, and Y. Cheng, "InfiniCache: Exploiting ephemeral serverless functions to build a cost-effective memory cache," in *Proc. Conf. on File and Storage Technologies (FAST'20)*, Feb. 2020, pp. 267–281.

[3] J. Yang, Y. Yue, and K. V. Rashmi, "A Large-Scale Analysis of Hundreds of In-Memory Key-Value Cache Clusters at Twitter," *ACM Transactions on Storage*, vol. 17, no. 3, pp. 1–35, Aug. 2021.

[4] J. Yang, Y. Yue, and R. Vinayak, "Segcache: A memory-efficient and scalable in-memory key-value cache for small objects," in *Proc. Symp. on Networked Systems Design and Implementation (NSDI'21)*, Apr. 2021, pp. 503–518.

[5] Y. Cheng, A. Gupta, and A. R. Butt, "An in-memory object caching framework with adaptive load balancing," in *Proc. of the European Conf. on Computer Systems (EuroSys'15)*, 2015, pp. 1–16.

[6] J. Kwak, E. Hwang, T.-K. Yoo, B. Nam, and Y.-R. Choi, "In-memory caching orchestration for Hadoop," in *Proc. Intl. Symp. on Cluster, Cloud and Grid Computing (CCGrid'16)*, May 2016, pp. 94–97.

[7] Redis Labs, "Redis," https://redis.io.

[8] Memcached, "Memcached," https://memcached.org.

[9] Amazon, "Amazon Web Services," https://aws.amazon.com.

[10] Google, "Google Cloud," https://cloud.google.com.

[11] Microsoft, "Microsoft Azure," https://azure.microsoft.com.

[12] IBM, "IBM Cloud," https://www.ibm.com/cloud.

[13] T. Zhu, A. Gandhi, M. Harchol-Balter, and M. A. Kozuch, "Saving cash by using less cache," in *Proc. Workshop on Hot Topics in Cloud Computing (HotCloud'12)*, Jun. 2012.

[14] R. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[15] F. Olken, "Efficient Methods for Calculating the Success Function of Fixed-Space Replacement Policies," Tech. Rep. LBL-12370, May 1981.

[16] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan, "PARDA: A fast parallel reuse distance analysis algorithm," in *Proc. Intl. Parallel and Distributed Processing Symp. (IPDPS'12)*, Aug. 2012, pp. 1284–1294.

[17] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, "Efficient MRC construction with SHARDS," in *Proc. Conf. on File and Storage Technologies (FAST'15)*, Feb. 2015, pp. 95–110.

[18] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, "Cache modeling and optimization using Miniature Simulations," in *Proc. USENIX Annual Technical Conf. (USENIX ATC'17)*, Jul. 2017, pp. 487–498.

[19] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, and Z. Wang, "Kinetic modeling of data eviction in cache," in *Proc. USENIX Annual Technical Conf. (USENIX ATC'16)*, Jun. 2016, pp. 351–364.

[20] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield, "Characterizing storage workloads with Counter Stacks," in *Proc. Symp. on Operating Systems Design and Implementation (OSDI'14)*, Oct. 2014, pp. 335–349.

[21] B. T. Bennett and V. J. Kruskal, "LRU Stack Processing," *IBM Journal of Research and Development*, vol. 19, no. 4, pp. 353–357, 1975.

[22] D. Carra and G. Neglia, "Efficient miss ratio curve computation for heterogeneous content popularity," in *Proc. USENIX Annual Technical Conf. (USENIX ATC'20)*, Jul. 2020, pp. 741–751.

[23] D. Narayanan, A. Donnelly, and A. Rowstron, "Write Off-Loading: Practical Power Management for Enterprise Storage," *ACM Transactions on Storage*, vol. 4, no. 3, pp. 1–23, Nov. 2008.

[24] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *Proc. Conf. of the IEEE Computer and Communications Societies (INFOCOM'99)*, Mar. 1999, pp. 126–134.

[25] G. Hasslinger, J. Heikkinen, K. Ntougias, F. Hasslinger, and O. Hohlfeld, "Optimum caching versus LRU and LFU: Comparison and combined limited look-ahead strategies," in *Proc. Intl. Symp. on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt'18)*, May 2018, pp. 1–6.

[26] O. Eytan, D. Harnik, E. Ofer, R. Friedman, and R. Kat, "It's time to revisit LRU vs. FIFO," in *Proc. Workshop on Hot Topics in Storage and File Systems (HotStorage'20)*, Jul. 2020.

[27] S. Min, D. Lee, C. Kim, J. Choi, J. Kim, Y. Cho, and S. Noh, "LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies," *IEEE Transactions on Computers*, vol. 50, no. 12, pp. 1352–1361, Dec. 2001.

[28] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, M. Tan *et al.*, "Semantic data caching and replacement," in *Proc. Intl. Conf. on Very Large Data Bases (VLDB'96)*, Sep. 1996, pp. 330–341.

[29] T. Johnson, D. Shasha *et al.*, "2Q: A low overhead high performance buffer management replacement algorithm," in *Proc. Intl. Conf. on Very Large Data Bases (VLDB'94)*, Sep. 1994, pp. 439–450.

[30] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson, "Dynamic performance profiling of cloud caches," in *Proc. Symp. on Cloud Computing (SOCC'14)*, Nov. 2014, pp. 1–14.

[31] B. Reed and D. D. E. Long, "Analysis of Caching Algorithms for Distributed File Systems," *SIGOPS Operating Systems Review*, vol. 30, no. 3, pp. 12–21, Jul. 1996.

[32] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Cliffhanger: Scaling performance cliffs in web memory caches," in *Proc. Symp. on Networked Systems Design and Implementation (NSDI'16)*, Mar. 2016, pp. 379–392.

[33] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman, "Memshare: A dynamic multi-tenant key-value cache," in *Proc. USENIX Annual Technical Conf. (USENIX ATC'17)*, Jul. 2017, pp. 321–334.

[34] J. Yang, Y. Zhang, Z. Qiu, Y. Yue, and R. Vinayak, "FIFO queues are all you need for cache eviction," in *Proc. Symp. on Operating Systems Principles (SOSP'23)*, Oct. 2023, pp. 130–149.

[35] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. Conf. on File and Storage Technologies (FAST'03)*, Mar. 2003, pp. 115–130.

[36] S. Sultan, K. Shakiba, A. Lee, P. Chen, and M. Stumm, "TTLs matter: Efficient cache sizing with TTL-aware miss ratio curves and working set sizes," Apr. 2024, submitted to Proc. of the European Conf. on Computer Systems (EuroSys'24).

[37] U.S. Securities and Exchange Commission (SEC), "Edgar log file data sets," https://www.sec.gov/about/data/edgar-log-file-data-sets.

[38] J. Ryans, "Using the EDGAR log file data set," 2017. [Online]. Available: https://dx.doi.org/10.2139/ssrn.2913612

[39] Twitter, "Github: twitter/cache-trace," https://github.com/twitter/cache-trace.

[40] X. Hu, X. Wang, L. Zhou, Y. Luo, Z. Wang, C. Ding, and C. Ye, "Fast Miss Ratio Curve Modeling for Storage Cache," *ACM Transactions on Storage*, vol. 14, no. 2, pp. 1–34, Apr. 2018.

[41] D. Matani, K. Shah, and A. Mitra, "An O(1) Algorithm for Implementing the LFU Cache Eviction Scheme," *arXiv preprint arXiv:2110.11602*, Oct. 2021.

[42] The kernel development community, "The /proc filesystem," https://www.kernel.org/doc/html/latest/filesystems/proc.html.

[43] X. Gu and C. Ding, "On the theory and potential of LRU-MRU collaborative cache management," in *Proc. Intl. Symp. on Memory Management (ISMM'11)*, Jun. 2011, pp. 43–54.

[44] A. Yu, Y. Tan, C. Xu, Z. Ma, D. Liu, and X. Chen, "DFShards: Effective construction of MRCs online for non-stack algorithms," in *Proc. Intl. Conf. on Computing Frontiers (CF'21)*, May 2021, pp. 63–72.

[45] X. Gu and C. Ding, "A generalized theory of collaborative caching," in *Proc. Intl. Symp. on Memory Management (ISMM'12)*, Jun. 2012, pp. 109–120.

[46] N. Beckmann and D. Sanchez, "Talus: A simple way to remove cliffs in cache performance," in *Proc. Intl. Symp. on High Performance Computer Architecture (HPCA'15)*, Feb. 2015, pp. 64–75.

[47] J. Alghazo, A. Akaaboune, and N. Botros, "SF-LRU cache replacement algorithm," in *Proc. Workshop on Memory Technology, Design and Testing*, Aug. 2004, pp. 19–24.

[48] P. Ranjan Panda, H. Nakamura, N. Dutt, and A. Nicolau, "A data alignment technique for improving cache performance," in *Proc. Intl. Conf. on Computer Design VLSI in Computers and Processors (ICCD'97)*, Oct. 1997, pp. 587–592.

[49] D. Thiebaut, J. Wolf, and H. Stone, "Improving Disk Cache Hit-Ratios Through Cache Partitioning," *IEEE Transactions on Computers*, vol. 41, no. 06, pp. 665–676, Jun. 1992.

[50] N. Beckmann, H. Chen, and A. Cidon, "LHD: Improving cache hit rate by maximizing hit density," in *Proc. Symp. on Networked Systems Design and Implementation (NSDI'18)*, Apr. 2018, pp. 389–403.

[51] A. Blankstein, S. Sen, and M. J. Freedman, "Hyperbolic caching: Flexible caching for web applications," in *Proc. USENIX Annual Technical Conf. (USENIX ATC'17)*, Jul. 2017, pp. 499–511.

[52] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang, "LAMA: Optimized locality-aware memory allocation for key-value cache," in *Proc. USENIX Annual Technical Conf. (USENIX ATC'15)*, Jul. 2015, pp. 57–69.

[53] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *Proc. Intl. Symp. on Microarchitecture (MICRO'12)*, Dec. 2012, pp. 389–400.

[54] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu, "MEMTUNE: Dynamic memory management for in-memory data analytic platforms," in *Proc. Intl. Parallel and Distributed Processing Symp. (IPDPS'16)*, May 2016, pp. 383–392.

[55] A. Nasu, K. Yoneo, M. Okita, and F. Ino, "Transparent in-memory cache management in Apache Spark based on post-mortem analysis," in *Proc. Intl. Conf. on Big Data (Big Data'19)*, Dec. 2019, pp. 3388–3396.

[56] M. Bilal and S.-G. Kang, "Time aware least recent used (TLRU) cache management policy in ICN," in *Proc. Intl. Conf. on Advanced Communication Technology (ICACT'14)*, Feb. 2014, pp. 528–532.

[57] G. Einziger, R. Friedman, and B. Manes, "TinyLFU: A Highly Efficient Cache Admission Policy," *ACM Transactions on Storage*, vol. 13, no. 4, pp. 1–31, Nov. 2017.

[58] T. B. G. Perez, X. Zhou, and D. Cheng, "Reference-distance eviction and prefetching for cache management in Spark," in *Proc. Conf. on Parallel Processing (ICPP'18)*, 2018, pp. 1–10.

[59] M. Takagi and K. Hiraki, "Inter-reference gap distribution replacement: An improved replacement algorithm for set-associative caches," in *Proc. Intl. Conf. on Supercomputing (ICS'04)*, Jun. 2004, pp. 20–30.

[60] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," in *Proc. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'90)*, Apr. 1990, pp. 134–142.

[61] N. Beckmann and D. Sanchez, "Modeling cache performance beyond LRU," in *Proc. Intl. Symp. on High Performance Computer Architecture (HPCA'16)*, Mar. 2016, pp. 225–236.

[62] E. Berg and E. Hagersten, "StatCache: A probabilistic approach to efficient and accurate data locality analysis," in *Proc. Intl. Symp. on Performance Analysis of Systems and Software (ISPASS'04)*, Mar. 2004, pp. 20–27.

[63] D. Eklov and E. Hagersten, "StatStack: Efficient modeling of LRU caches," in *Proc. Intl. Symp. on Performance Analysis of Systems Software (ISPASS'10)*, Mar. 2010, pp. 55–65.

[64] D. Thiebaut, “On the Fractal Dimension of Computer Programs and its Application to the Prediction of the Cache Miss Ratio,” *IEEE Transactions on Computers*, vol. 38, no. 7, pp. 1012–1026, Jul. 1989.

[65] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, “Dynamic tracking of page miss ratio curve for memory management,” in *Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS’04)*, Oct. 2004, pp. 177–188.

[66] D. Byrne, N. Onder, and Z. Wang, “mPart: Miss-ratio curve guided partitioning in key-value stores,” in *Proc. Intl. Symp. on Memory Management (ISMM’18)*, Jun. 2018, pp. 84–95.

[67] M. K. Qureshi and Y. N. Patt, “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches,” in *Proc. Intl. Symp. on Microarchitecture (MICRO’06)*, Dec. 2006, pp. 423–432.

[68] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, “Dynacache: Dynamic cloud caching,” in *Proc. Workshop on Hot Topics in Cloud Computing (HotCloud’15)*, Jul. 2015.

[69] Z. Liu, H. W. Lee, Y. Xiang, D. Grunwald, and S. Ha, “eMRC: Efficient miss ratio approximation for multi-tier caching,” in *Proc. Conf. on File and Storage Technologies (FAST’21)*, Feb. 2021, pp. 293–306.

[70] S. Jiang and X. Zhang, “LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance,” in *Proc. Intl. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS’02)*, Jun. 2002, pp. 31–42.

I/O Passthru: Upstreaming a flexible and efficient I/O Path in Linux

Kanchan Joshi[1], Anuj Gupta[1], Javier González[1], Ankit Kumar[1], Krishna Kanth Reddy[1], Arun George[1], Simon Lund[1], and Jens Axboe[2]

[1]Samsung Semiconductor
[2]Meta Platforms Inc

Abstract

New storage interfaces continue to emerge fast on Non-Volatile Memory Express (NVMe) storage. Fitting these innovations in the general-purpose I/O stack of operating systems has been challenging and time-consuming. The NVMe standard is no longer limited to block-I/O, but the Linux I/O advances historically centered around the block-I/O path. The lack of scalable OS interfaces risks the adoption of the new storage innovations.

We introduce *I/O Passthru*, a new I/O Path that has made its way into the mainline Linux Kernel. The key ingredients of this new path are *NVMe char interface* and *io_uring command*. In this paper, we present our experience building and upstreaming I/O Passthru and report on how this helps to consume new NVMe innovations without changes to the Linux kernel. We provide experimental results to (i) compare its efficiency against existing *io_uring* block path and (ii) demonstrate its flexibility by integrating data placement into Cachelib. FIO peak performance workloads show 16-40% higher IOPS than block path.

1 Introduction

The Non-Volatile Memory Express (NVMe) protocol has been the unquestionable catalyst for the broad adoption of NAND-based storage devices and Solid-State Drives (SSDs). NVMe continues to bring new capabilities in terms of performance and functionality. Low latency, high bandwidth SSDs (such as Intel optane [61], Kioxia's FL6 [11], Samsung's ZNAND [25]) use NVMe as the protocol of choice. Functionality expansion comes from new commands and command sets that make NVMe viable for unconventional block storage. In the past few years, several non-block storage interfaces have gained popularity and, lately, standardization in NVMe. Specifically, in data-placement solutions, Open-Channel SSDs [37, 44, 47] gained popularity in the academia and industry and eventually opened the door for the standardization of Zoned Namespaces (ZNS) in NVMe. As of today, NVMe standardizes several new interfaces, including Multi-Stream (NVMe Directives) [19], Key-Value [19], Zoned Namespaces [19], and Flexible Data Placement [17]; more interfaces such as Computational Storage [23] are still under development.

It is relevant to note that all of these new interfaces require vertical integration across different storage stack layers (driver, block-layer, file systems) and define new user interfaces to accommodate new device interfaces. Such changes are not always welcomed, as they go against the principle of maintaining a stable and general-purpose operating system. Linux Kernel goes to great lengths to abstract the hardware and never breaks the user-space. This presents a difficult trade-off as robustness and maintenance of the operating system lock horns with early enablement and adoption of NVMe innovations.

In this paper, we present *I/O Passthru*, a novel I/O path in mainline Linux kernel that (i) allows the deployment of any new NVMe feature much faster as it is devoid of extra abstractions and (ii) provides an efficient and feature-rich user-interface. To summarize, our main contributions are the following:

- We build a new NVMe passthrough I/O path which provides higher flexibility and efficiency than the block I/O path (Section 4). We provide examples of how this path enables NVMe interfaces, such as flexible data placement, computational storage, and end-to-end data protection (Section 6).
- We introduce io_uring command, a generic facility to implement asynchronous IOCTLs in the Linux kernel. We detail its API and design (Section 4.2.1).
- We get this path upstream in the Linux Kernel (Section 5) and integrate it into user-space software, including SPDK, xNVMe, liburing, fio and nvme-cli (Section 5.2).
- We elaborate on factors that influence the efficiency of I/O and evaluate the proposed path. FIO peak-performance workloads show 16-40% higher IOPS (Section 7.1).

2 Motivation and Background

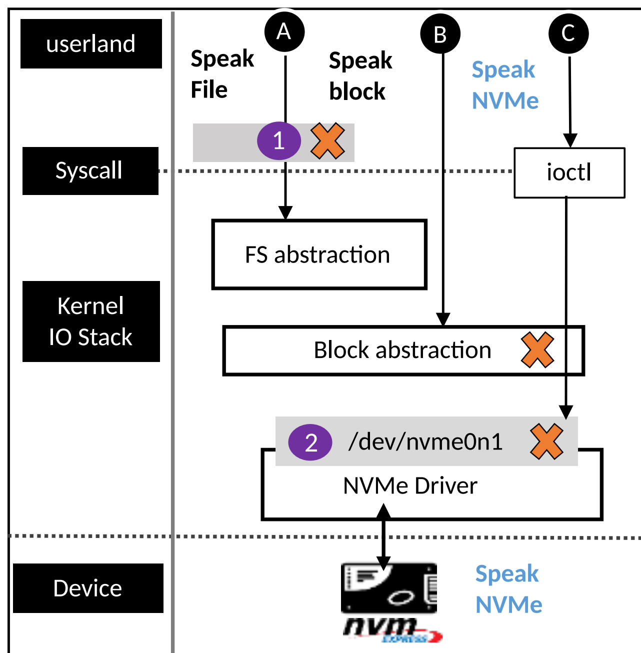


Figure 1: Abstraction layers across different I/O paths and its failures undermining usability, availability and efficiency

2.1 NVMe innovations vs Kernel abstractions

The primary motivation is the fast-paced growth of NVMe innovations and the Linux kernel's agility, or lack thereof, to consume those. **NVMe, initially meant to support only block storage, is no longer tied to it.** This is possible after the introduction of an entity named command-set in NVMe standard [19]. NVMe 2.0 specification defines three command sets:

- NVM command set corresponds to block storage. Nevertheless, it continues to grow newer ways of interacting with storage. For example, (i) data-placement methods like multi-stream and FDP [17] involve passing hints with write, (ii) copy command that does not involve host buffers and performs in-device copy instead.
- Zoned namespace command set exposes zones to the Host and presents new I/O commands such as zone-append and zone-management send/receive.
- Key-value command set does away with fixed-size logical blocks and speaks keys/values instead.

Moreover, new command sets for computational-storage are shaping up. Command sets convey the divorce of NVMe from block-only storage, thereby ensuring faster future innovation.

As for the Linux kernel, generic abstractions are at the foundation. Figure 1 briefly describes various I/O paths in the Linux kernel. NVMe driver collaborates with the block layer, abstracts NVMe protocol, and presents a block device `/dev/nvme0n1` to the upper layer. This block device interface helps the file system to be NVMe agnostic. For example, the file system sends a write operation to the block device by forming a `bio` with `REQ_OP_WRITE`, which is translated to a protocol-specific write command by the underlying driver. The block interface is the bedrock that file systems use to create file abstraction. File systems collaborate with VFS to provide specific implementations of certain user-space APIs that are invoked as system calls. For the syscall users, the file system itself is abstracted. This is shown as path (A) in the figure. Path (B) is a subset when the block device is operated directly without any file system. Figure 1 outlines specific problems with the existing I/O paths:

- Many **new NVMe commands do not fit into the existing user interface.** Adding a new system call requires a more generic use case than the NVMe-specific one. Furthermore, a new syscall is discouraged as it has to be supported indefinitely [1]. Consequently, there is an increase in NVMe interfaces that still need a user interface in Linux. For example, zone-append [35], a variant of nameless writes [27] tailored for zoned storage which is supported by the block layer for in-kernel users, but lacks a user-space API due to the unconventional semantics. Also, while we count on several mentioned technologies to improve in-device data-placement decisions, we still do not have a streamlined way to communicate placement information with the existing write APIs. Finally, despite several efforts to open-source support for copy-offload [43, 54] given the existing hardware support [38, 45], we are yet to see mainline support with a matching user interface.
- One way to alleviate the user-interface scarcity is by using the NVMe passthrough path, shown as (C) in the figure. This path is devoid of file/block abstractions. Applications can send the NVMe command using the `ioctl` syscall [5]. However, this path comes at the **cost of efficiency**, as `ioctl` is a synchronous operation and not a good fit, particularly for the highly parallel NVMe storage. Apart from blocking nature, `ioctl` (and therefore passthrough I/O path) is far from various advancements (outlined in the Section 2.2) that have gone into regular read/write I/O path.
- All three paths (A), (B), and (C) rely on the block interface. However, **availability of block-interface is not guaranteed**. There are various situations when the block interface goes haywire. For example, (i) if a namespace is configured to transfer data and metadata as extended LBA(logical block address), the block device is marked with zero-capacity, which prevents further block/file I/O, (ii) ZNS device without zone-append is marked read-only, (iii) ZNS device with non-power of two zone-size is marked hidden, and (iv) any non-block command-set, e.g., Key-value, can not be operated with the block interface.

2.2 I/O advances with io_uring

io_uring is the latest and most feature-rich asynchronous I/O subsystem in Linux [29]. It operates at the boundary of user-space/kernel and covers storage and network I/O. The communication backbone is a pair of ring buffers, Submission Queue (SQ)/Completion Queue (CQ), shared between user-space and kernel. The application creates these rings by calling `io_uring_setup` call. It prepares the I/O by extracting an entry from SQ called `SQE`. It fills up the SQE and submits the I/O by calling `io_uring_enter` system call [6]. Finally, it obtains the completion by extracting an `CQE` entry from CQ. io_uring brings various advancements in the I/O path, some of which are outlined below:

- **Batching:**Allow submission of multiple I/O requests in one shot with a single system call.
- **SQPoll:**Syscall-free submissions. The application can offload the submission of I/O to a kernel thread that io_uring creates.
- **IOPoll:**Completion can be polled by setting IORING_SETUP_IOPOLL flag on the ring. This gives interrupt-free I/O.
- **Chaining:**Allows to establish ordering/dependencies among multiple commands at the time of submission. For example, write followed by a read (i.e., copy semantics) and commonly used sequence open-read-close. This is possible by chaining adjacent SQEs with the flag IOSQE_IO_LINK and submitting the entire chain in a go with a single syscall.

Ever since its inception, io_uring has added async variants of various sync system calls [52]. Two methods for turning a sync operation into async are outlined below.

- **Worker-based async**: spawn a worker thread and delegate sync operation to it. The advantage is the low implementation effort. However, this causes overheads and does not scale.
- **True-async**: fast and scalable as it does not involve worker threads. It relies on ensuring that the submitter does not block during the submission. Implementation effort grows as all components participating in the operation (e.g., file system, driver) should provide wait-free compliance.

io_uring employs both methods depending on the operation. For more common read/write operations, it uses true-async and falls back to worker-based async if need be. For known blocking operations (e.g., mkdirat, fsync), worker-based async is used in the first place itself.

3 Design considerations

3.1 Limitations of existing NVMe passthrough

The upstream Linux NVMe driver presents a passthrough interface to applications using these ioctl-driven opcodes:

- `NVME_IOCTL_IO64_CMD` is used to send NVMe I/O commands.
- `NVME_IOCTL_ADMIN64_CMD` is used to send NVMe admin commands.

Both these ioctls operate on `struct nvme_passthru_cmd64`, shown in Listing 1.

```
struct nvme_passthru_cmd64 {
  __u8   opcode;
  __u8   flags;
  __u16 rsvd1;
  __u32 nsid;
  __u32 cdw2;
  __u32 cdw3;
  __u64 metadata;
  __u64 addr;
  __u32 metadata_len;
  union {
    __u32 data_len;
    __u32 vec_cnt;
  };
  __u32 cdw10;
  __u32 cdw11;
  __u32 cdw12;
  __u32 cdw13;
  __u32 cdw14;
  __u32 cdw15;
  __u32 timeout_ms;
  __u32   rsvd2;
  __u64 result;
};
```

Listing 1: control structure that user-space sends for sync passthorugh

User-space forms the command using this structure, which is 80 bytes in size. Upon submission, the NVMe driver copies this to kernel-space using `copy_from_user` operation. It maps the data (line 9) and metadata buffer (line 8) and eventually submits the NVMe command to the device. The caller is put to wait until completion arrives. On completion, the primary result is sent to the user-space using the ioctl return value, and another one is updated into the result field (line 23). For the latter, the driver does a `put_user` operation.

While this interface allows to bypass the abstractions, it suffers several limitations:

- It is tied to the block device, which itself is fragile.
- Ioctl, due to its blocking interface, harms both scalability and efficiency. Figure 3 shows that io_uring random read scales perfectly, while ioctl driven read stays flat.
- As the above sequence outlines, there is a per-command overhead of copying command and result between user and kernel space.
- This interface can only be used by the root user.

3.2 Design goals

Based on the shortcomings mentioned in Section 2.1 and 3.1, we set the main design goals as follows:

- **Block I/O independence**: The block interface cannot represent the non-block command sets that NVMe has.

The new interface should have higher flexibility and cover all NVMe command sets regardless of their non-block semantics.
- **Catch-all user interface**: Adding a new syscall in Linux every time NVMe gets a new command is impractical. For every existing and future NVMe command, the solution should ensure a user interface without coining it.
- **Efficient and scalable**: NVMe represents fast and parallel storage. The new interface should have the same or higher efficiency and scalability than the direct block I/O path.
- **General accessibility**: The solution should not only be locked to the root/admin user.
- **Upstream acceptance**: The solution should become part of the official Linux repositories. This ensures that adopters do not have to reinvent or maintain off-tree code.

4 I/O Passthru in Kernel: Architecture and Implementation

The proposed I/O path is shown in Figure 2, with the label (D). It consists of a new char-interface `/dev/ng0n1` as the backend, which interfaces with io_uring using newly introduced `io_uring_command`.

We also considered Linux AIO but chose io_uring for two reasons. First, it is more efficient and feature-rich, as outlined in Section 2.2. Second, it is a more active subsystem in the upstream Linux kernel. The following sections detail the design and implementation by grouping those into three attributes - (i) availability, (ii) efficiency, and (iii) accessibility.

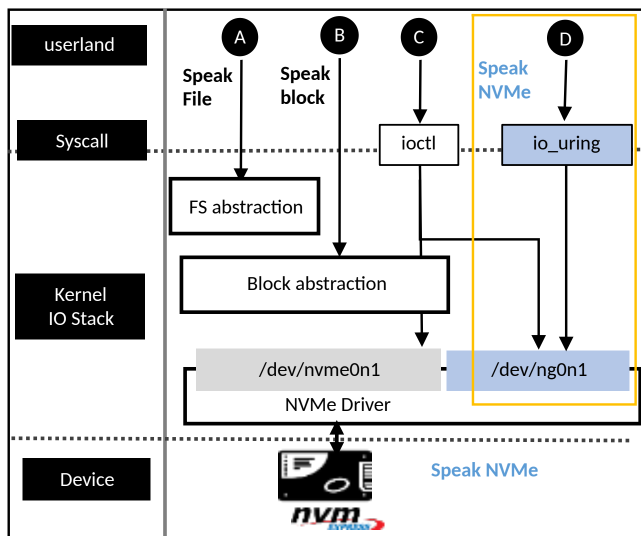


Figure 2: New passthrough I/O path, marked with (D) and enclosed with the dotted rectangle.

4.1 Availability: NVMe generic char interface

NVMe generic device solves the availability problem associated with the block device. We modify the NVMe driver to create a character device node for each namespace found on the NVMe device, and more importantly, this is done regardless of any unsupported feature that may break the block device (Section 2.1). Char device is also created for unknown command sets, e.g., anything other than NVM and ZNS. Therefore, the char interface is bound to appear for future command sets without requiring any further code changes to the NVMe driver. While the block device follows the naming convention `/dev/nvme<X>n<Y>`, the char device follows `/dev/ng<X>n<Y>`. The term `ng` refers to NVMe-generic as it applies to any NVMe command set. Listing 2 shows the `file_operations` for this character device. User-space can send any NVMe command through the character device using `ioctl`. Line 6 shows the `ioctl` handler in the NVMe driver. More importantly, the NVMe char device also talks to io_uring to enable a bunch of advances. This is shown in line 8 via the `uring_cmd` handler and elaborated in the next section.

```
1  const struct file_operations nvme_ns_chr_fops =
2  {
3    .owner            = THIS_MODULE,
4    .open             = nvme_ns_chr_open,
5    .release          = nvme_ns_chr_release,
6    .unlocked_ioctl = nvme_ns_chr_ioctl,
7    .compat_ioctl    = compat_ptr_ioctl,
8    .uring_cmd       = nvme_ns_chr_uring_cmd,
9    .uring_cmd_iopoll = nvme_ns_chr_uring_cmd_iopoll
       ,
10 };
```

Listing 2: `file_operations` for NVMe char device

4.2 Infusing the efficiency & scalability

Solving the efficiency limitation of NVMe passthrough requires solving the more fundamental problem in Linux - coining an efficient alternative of ioctl. This alternative must be generic enough to be applied beyond the NVMe use case. To that end, we added three new facilities in io_uring:io_uring command, Big SQE, and Big CQE. Then, we outline how we employ these facilities to construct a new NVMe passthrough path. To further reduce the per I/O overhead, we wire up two more capabilities: fixed-buffer and completion-polling.

4.2.1 io_uring command

A relatively simple way to introduce ioctl-like capability in io_uring is to use the worker-based-async approach (Section 2.2). However, that will be anything but scalable. Figure 3 shows io_uring scaling for 512b random-read with and without the worker thread. With the default true-async approach, throughput soars as queue depth increases and reaches 3.5M IOPS. However, with the worker approach, the throughput

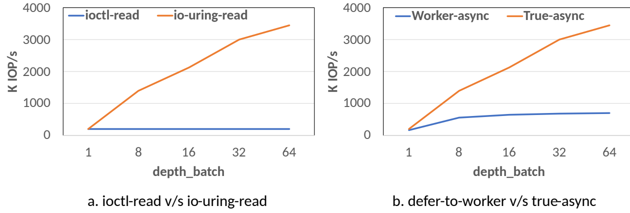


Figure 3: Performance comparison

does not increase beyond 500K. Therefore, we go by the true-async design approach to add this new facility named `io_uring command`. User interface involves preparing `SQE` with a new operation code `IORING_OP_URING_CMD`. The command is to be placed inline within the `SQE`. This relieves the application from the command allocation and provides zero-copy communication as `SQE` is shared between user and kernel space. Regular `SQE` has 16 bytes of free space that the application can use for housing the command. The application gets to reap the result from the `CQE`. Regular `CQE` provides a signed 4-byte value as the result.

Big SQE: Regular SQE with 16 bytes of free space is not enough as the NVMe passthrough command is about 80 bytes in size (listing 1). Therefore, we introduce the facility to create the ring with a larger SQE. `Big SQE` is double the size of regular `SQE` and provides 80 bytes of free space. The application can set up the ring with `Big SQE` by specifying the flag `IORING_SETUP_SQE_128`.

Big CQE: Some NVMe commands return more than one result to the user-space. For example, the zone-append command returns the location where the write landed. And io_uring regular CQE lacks the ability to return more than one result. To tackle that, we introduce `Big CQE`, which is double the size of regular `CQE` and provides 16 bytes of extra space to return additional information to user-space. The flag `IORING_SETUP_CQE_32` allows the application to set up the ring with `Big CQE`.

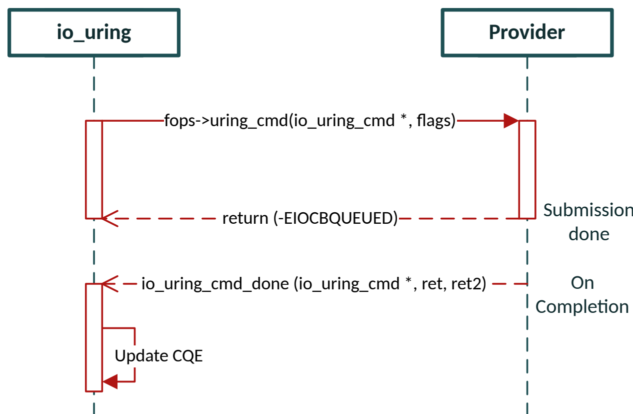


Figure 4: `uring_cmd` communication flow overview

We implemented `io_uring command` to be generic to support any underlying command. The **command provider** can be any kernel component (e.g., file system, driver) that collaborates with io_uring. While the NVMe driver is the first command-provider that got into the kernel, other examples include ublk [48] and network sockets [49]. The communication between io_uring and command-provider follows the true-async design approach (Section 2.2), and this is outlined in Figure 4. During the submission, io_uring processes the `SQE` and prepares another `struct io_uring_cmd` (Listing 3) that is used for all further communication. io_uring invokes the command-provider by `->uring_cmd` handler of file_operations. The provider does what is necessary for the submission and returns to io_uring without blocking. Actual completion is decoupled from submission and is rather done when the provider calls `io_uring_cmd_done` with the primary and auxiliary result. The primary result is placed into regular `CQE`, and the auxiliary result goes to `Big CQE`.

```
struct io_uring_cmd {
  struct file   *file;
  const void    *cmd;
  union {
    /* to defer completions to task context */
    void (*task_work_cb) (struct io_uring_cmd *cmd
    );
    /* for polled completion */
    void *cookie;
  };
  u32 cmd_op;
  u32 flags;
  u8 pdu[32]; /* available inline for free use */
};
```

Listing 3: `struct io_uring_cmd` for in-kernel communication

4.2.2 Asynchronous processing

For the new io_uring driven passthrough we add the following opcodes in NVMe driver:

- `NVME_URING_CMD_IO` : for NVMe I/O commands.
- `NVME_URING_CMD_IO_VEC`: vectored variant of the above.
- `NVME_URING_CMD_ADMIN`: for NVMe admin commands.
- `NVME_URING_CMD_ADMIN_VEC`: vectored variant of the above.

Vectored variants. Allow multiple data buffers to be passed from user-space, similar to what is possible for classical I/O using readv/writev syscalls. The above four operations expect a new `struct nvme_uring_cmd` as input. This is shown in Listing 4.

```
struct nvme_uring_cmd {
    __u8  opcode;
    __u8  flags;
    __u16 rsvd1;
    __u32 nsid;
    __u32 cdw2;
```

```
    __u32 cdw3;
    __u64 metadata;
    __u64 addr;
    __u32 metadata_len;
    __u32 data_len;
    __u32 cdw10;
    __u32 cdw11;
    __u32 cdw12;
    __u32 cdw13;
    __u32 cdw14;
    __u32 cdw15;
    __u32 timeout_ms;
    __u32   rsvd2;
};
```

Listing 4: control structure that user-space sends for uring passthorugh

Zero copy. User-space creates this structure within the Big SQE itself, eliminating the need for copy_from_user. Also, the auxiliary result is returned via Big CQE, so put_user is avoided. Therefore, this structure does not have a result field embedded into it. This ensures zero-copy in the control path. **Zero memory-allocations.** Unlike sync passthrough, we ensure that command completion is decoupled from submission and the submitter is not blocked. This asynchronous processing requires some fields to be persistent (until completion), so these fields cannot be created on the stack. Dynamically allocating these fields will add to the latency of I/O. We avoid dynamic allocation by reusing the free space pdu inside the struct io_uring_cmd (Listing 3, line 12) for such bookkeeping.

4.2.3 Fixed-buffer

I/O buffers must be locked into the memory for any data transfer. This adds to the per I/O cost as buffers are pinned and unpinned during the operation. However, this can be optimized if the same buffers are used for I/O repeatedly. Therefore, io_uring can pin several buffers upfront using io_uring_register. The application can use these buffers for I/O using opcodes such as IORING_OP_READ_FIXED or IORING_OP_WRITE_FIXED.

We introduce this capability for uring_cmd using a new flag IORING_URING_CMD_FIXED instead. The application specifies this flag and buffer index in the SQE. Within the kernel, the NVMe driver checks the presence of this flag. If found set, it does not attempt to lock the buffer. Instead, it talks to io_uring to reuse the previously locked region. To that end, we add a new in-kernel API io_uring_cmd_import_fixed that any command provider can use.

4.2.4 Completion polling

io_uring allows the application to do interrupt-free completions for read/write I/O. This helps in reducing the context-switching overhead as the application engages in active polling rather than relying on the interrupts. NVMe driver, when loaded with polled_queues = N parameter, sets up N polled queue-pair (SQ and CQ) for which NVMe device does not generate the interrupt on command-completion. Since io_uring decouples submission from completion, async polling for completion is possible. This is more useful than sync polling, as the application can do other work rather than spinning on the CPU just after a single submission.

We extend async polling for uring_cmd too. For this, two things are done differently during submission in the NVMe driver - (i) polled-queue is chosen for command submission, (ii) a submission identifier cookie is stored in struct io_uring_cmd (line 8, Listing 3). Two identifiers are required to pinpoint the particular command during completion: (i) queue-identifier, in which the command is submitted, and (ii) command-identifier within that queue. These two identifiers are combined into a single 4-byte entity referred to as cookie.

For completion, a new callback uring_cmd_iopoll (line 9, listing 2) is added that implements the polling loop for matching completion. It extracts the cookie from struct io_uring_cmd and uses that to look for the matching completion entry in the NVMe completion queue.

4.3 Accessibility: from root-only to general

Linux uses discretionary access control (DAC) as the default way to manage object access. File mode is a numeric representation that specifies who (file owner, member of a group, or anyone else) is allowed to do what (read, write, or execute).

The VFS uses file mode to do the first level of permission checks when the application requests to open the file. The second level of check is done by the NVMe driver when the application issues the command using the opened file handle. However, the NVMe driver guards all passthrough operations by a coarse-granular CAP_SYS_ADMIN check that disregards the file mode completely.

Listing 5 shows an example in which ng0n1 has a less restrictive file mode, i.e., 0666, compared to ng0n2.

```
$ ls -l --time-style=+ /dev/ng*
crw-rw-rw- 1 root root 242, 0 /dev/ng0n1
crw------- 1 root root 242, 1 /dev/ng0n2
```

Listing 5: example file-mode for char device

Even though ng0n1 has been marked to allow unprivileged read/write operations, nothing goes through. Instead, it behaves the same as ng0n2. The all-or-nothing CAP_SYS_ADMIN check renders the passthrough interface limited to the root user.

We modify the NVMe driver to implement a fine-granular policy that takes file-mode and command type into account for access control. This policy is defined as follows:

- When CAP_SYS_ADMIN is present, everything is allowed as before. Otherwise, the command type (admin command or I/O command) is checked.

- Any I/O command that can write/alter the device is only allowed if file-mode contains write permission.
- Any I/O command that only reads/obtains the information from the device is allowed.
- Admin commands such as identify-namespace and identify-controller are allowed. This is because these commands provide information that is necessary to form the I/O command. Other admin commands are not allowed.

Beyond DAC, the `uring_cmd` also supports mandatory access control (MAC). A new Linux Security Module (LSM) hook is defined for `uring_cmd` and SELinux [15] and Smack [16] implement the respective policy for the hook.

4.4 Block layer: To bypass or not

Does NVMe passthrough mean bypassing the block layer? It is a common misconception. Passthrough is rather about not placing another layering over the device. The NVMe generic char-device, introduced in this work, does away with the block abstraction altogether and presents cleaner semantics than passthrough over the block-device. Figure 5 shows how I/O Passthru interacts with the block layer during the submission. The block layer implements many common functionalities,

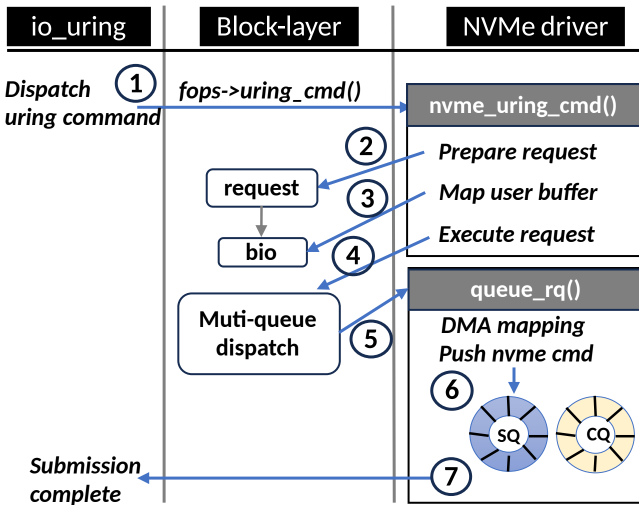


Figure 5: Integration with the block layer

either entirely or in collaboration with the underlying storage driver. Bypassing the block layer is not practical as it requires either reinventing or giving away the functionalities, turning the passthrough toothless. Table 1 presents the comparison.

- **Abstract device limits**. For example, the block layer makes it possible to send larger read I/O on a device that does not support single read commands to be larger than 64KB. For this, the block layer splits the larger read into many 64KB commands. Passthrough, by definition, does not abstract the device limits.

Feature	Block I/O	Passthrough I/O
`Abstract device limits`	Yes	No
`Scheduler bypass`	No	Yes
`Core to queue mapping`	Yes	Yes
`Command tags mgmt.`	Yes	Yes
`Timeout value`	Global	Per I/O
`Abort`	Yes	Yes

Table 1: Block layer functionalities: Block & Passthrough path

- **I/O scheduler**. Since I/O schedulers can merge the incoming I/Os, they are skipped for passthrough I/O. This is not a spoilsport, as not using the I/O scheduler performs best on NVMe SSDs. Generally, NVMe SSDs have deep queues and employ good internal I/O scheduling to meet SLAs. Prior studies have shown that Linux I/O schedulers (BFQ, mq-deadline, kyber) add significant overheads (up to 50%) and hamper scalability [12, 57]. Enterprise Linux distributions such as RHEL and SLES keep 'none' as the default scheduler for NVMe.
- **Muti queue**. Block-layer abstracts the device queues within the Blk-MQ infrastructure [36] and enables those to be shared among available cores. Passthrough also leverages this infrastructure.
- **Tag management**. The block layer manages the outstanding commands for each hardware queue. It manages the allocation/freeing of command IDs (tags) so that the driver does not need to implement flow control.
- **Command-timeout & Abort**.If a command takes longer than expected, the block layer can detect the timeout and abort the outstanding command. Passthrough supports user-specified timeout value (Listing 4, line 18), while block-path uses a hard-coded value for timeout.

5 Upstream

5.1 Kernel I/O Passthru Support

Table 2 shows the upstream progression of the proposed I/O path. All the constituent parts have made it to the official Linux kernel repository [14].

Feature	Kernel
Char-interface: initial support	5.13
Char-interface: any command-set	6.0
io_uring command	5.19
uring-passthrough for NVMe	5.19
Efficiency knobs (polling, fixed-buffer)	6.1
Unprivileged access for passthrough	6.2

Table 2: Upstream progression in the Linux kernel

5.2 Userspace I/O Passthru Support

5.2.1 xNVMe integration

xNVMe [51] is a cross-platform user-space library aimed at providing I/O interface independence to applications. xNVMe API abstracts multiple synchronous and asynchronous backends, including io_uring, libaio, and spdk. Application coded using xNVMe API can seamlessly switch among xNVMe's backends. We extend xNVMe to support a new asynchronous backend named `io_uring_cmd`. This backend works with NVMe character device `/dev/ngXnY`.

5.2.2 SPDK integration

SPDK contains a block-device layer, `bdev`, that implements a consistent block-device API over various devices underneath. For example, `NVMe bdev` is based on the NVMe driver of SPDK. `AIO bdev` and `uring bdev` are other examples that are implemented over Linux aio and io_uring respectively. We add a new bdev `xNVMe` in SPDK (shown in Figure 6). This single bdev allows to switch among AIO, io_uring, and io_uring_cmd. This bdev became part of SPDK since release version 22.09 [24].

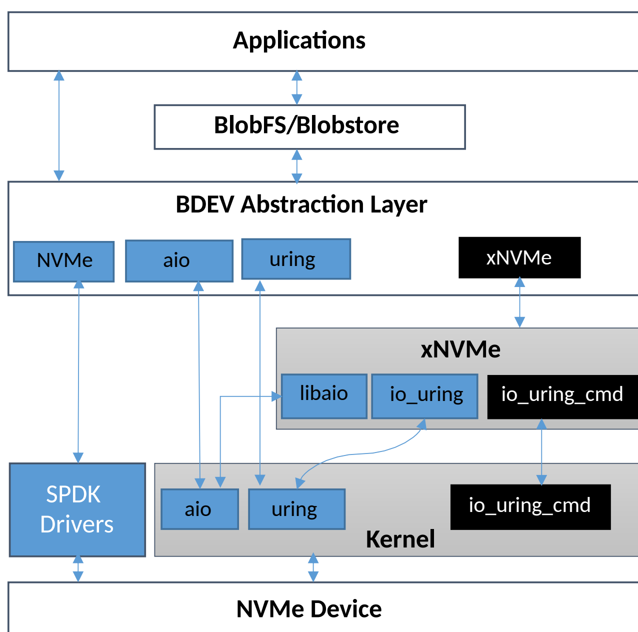


Figure 6: Overview of SPDK stack, and bdev_xnvme module

5.2.3 Tooling

nvme-cli [18] is modified to list character interface `/dev/ngXnY`. Any operation that nvme-cli can do on block-interface `/dev/nvmeXnY` can also be done on char-interface `/dev/ngXnY`.

Fio [30]: We add a new io-engine named `io_uring_cmd`. The user must pass a `cmd_type` when using this engine. This provides the flexibility to support other types of passthrough commands in the future. For NVMe passthrough, `cmd_type` is to be set as `nvme`, and the filename should be specified as `/dev/ngXnY`. This new ioengine is part of the Fio release since version 3.31. Fio repository contains a utility `t/io_uring` [32], which comes in handy to evaluate peak performance obtained via io_uring. We extend this utility so that io_uring NVMe passthrough can also be evaluated for peak performance.

Liburing [31] is the library that provides a simpler interface to io_uring applications. It is extended to support big-SQE and big-CQE. Moreover, we add a bunch of tests that issue uring-passthrough commands on the NVMe character device `/dev/ngXnY` [40].

6 Enabling NVMe interfaces with I/O Passthru

In this section, we present examples showing how the flexibility and efficiency of I/O Passthru help consume some NVMe features that are otherwise challenging to use in Linux.

6.1 Flexible Data Placement

Flexible data placement (FDP) is the latest host-guided data placement method in the NVMe standard. The ratified proposal [17] adds concepts such as reclaim unit (RU) and placement identifier (PID). RU is analogous to the SSD garbage-collection unit, and the host can place logical block addresses into RU by specifying PID in the write command. LBAs written with one-placement-identifier are not mixed with LBAs written with another placement-identifier. This helps to separate different data lifetimes and reduces write amplification in the SSD.

When multi-stream support was standardized in NVMe as directives, the Linux kernel developed the write-hint-based infrastructure that allowed applications to send the placement hints along with writes. However, this infrastructure is no longer functional as its core pieces have been purged from the mainline kernel [39]. I/O Passthru comes to the rescue as applications can send placement hints without worrying about vertical integration of FDP to various parts of the kernel storage stack. We demonstrate this with Cachelib, which can leverage FDP via I/O Passthru (Section 7.2). Also, FIO io_uring_cmd ioengine has supported FDP since version 3.34.

6.2 Computational Storage

Computational storage is a new architecture that allows the host to offload various compute operations to the storage, reducing data movement and energy consumption. The NVMe standardization is underway, and it involves presenting two new namespaces.

- **Memory namespace** refers to the subsystem-local-memory (SLM), a byte-addressable memory to enable the local processing of the SSD data. The host needs to issue new NVMe commands to (i) Transfer data between host-memory and SLM and (ii) Copy data between NVM namespace and SLM.
- **Compute namespace** represents various compute programs executed on the data residing in SLM. The host orchestrates the local data processing using a new set of NVMe commands: execute-program, load-program, activate-program, etc.

Supporting computational storage in Kernel is challenging because these new namespaces come with non-block semantics and various new unconventional commands. However, the generic char interface (`/dev/ngXnY`) comes up fine for both SLM and Compute namespace. All new NVMe commands can be issued efficiently with the I/O Passthru interface. Overall, this enables user-space to leverage computational storage without any changes to the Kernel.

6.3 End-to-End Data Protection

E2E data protection detects data integrity issues early and prevents corrupted data from being stored on the disk. Many NVMe SSDs have the ability to store extra metadata (8, 16, 32, 64 bytes) along with the data. This metadata can be interleaved with the data buffer (referred to as DIF) or in a separate buffer (referred to as DIX) [20]. This ability comes in handy to support erasure-coding, too. All or a portion (first or last bytes) of this metadata can contain protection information (PI) that contains checksum, reference tag, and application tag. The NVMe SSD controller verifies the PI contents while writing and reading.

Kernel support for E2E data protection [55] is limited, as shown in Figure 7. DIF is not supported as passing unaligned (e.g., 4096+8 bytes) data buffers is inconvenient. The block layer supports DIX as metadata is kept in a separate buffer. However, DIX is only supported if protection information resides in the first bytes of metadata. Also, PI is block-layer generated, and user-space applications cannot pass it due to a lack of interface.

I/O Passthru does not face buffer alignment checks or user-interface issues. The passthrough command structure allows applications to pass metadata buffer and length (Listing 4, lines 8 & 10). We have added DIF and DIX support in FIO io_uring_cmd ioengine.

7 Experiments

Table 3 summarizes our experimental setup. We conducted the experiments in three parts.

In the first part, we compare the efficiency of the new passthrough I/O path against the block I/O path on a direct-attached NVMe SSD. This is an apples-to-apples comparison

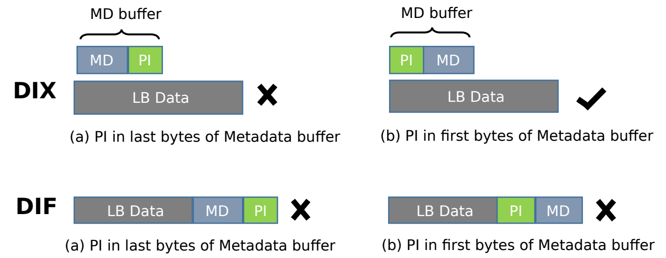


Figure 7: Block device limitations for DIF and DIX cases

between block interface `/dev/nvmeXnY` and char-interface `/dev/ngXnY`, as both are driven by io_uring. We exclude the sync passthrough path as it is known not to scale due to being `ioctl`-driven. We use Fio and t/io_uring utility, which is particularly suitable for peak-performance determination due to its low overhead. Both these are configured to run an unbuffered random read workload.

In the second part, we demonstrate the flexibility of the I/O Passthru interface by applying it in the real-world application Cachelib [3]. Cachelib is an open-source Caching engine from Meta which leverages RAM and SSD in the solution. Due to the nature of the workloads, Cachelib deployments can incur SSD Write Amplification (WAF > 2) on high SSD utilization scenarios. Therefore, the SSD utilization was limited to 50 percent in many production deployments. NVMe FDP tackles this problem of high WAF by segregating I/Os of different longevity types in the physical NAND media. We use the Samsung SSD that supports data placement using the NVMe FDP commands. Atop Cachelib, we run the production workload and compare the write-amplification against the case when FDP is not enabled.

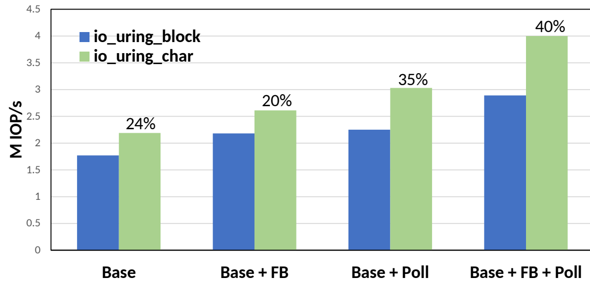
In the third part, we compare the scalability of block and passthrough I/O against the user-space SPDK NVMe driver.

Hardware	Model
CPU	AMD Ryzen 9 5900X 12-Core
Memory	DDR4 16 GB
Board	MSI MEG X570 GODLIKE
Storage	[1] Intel Optane P5800X, 400GB Spec: 5M (512b RR), 1.6M (4K RW) [2] Samsung FDP SSD, 7.5 TB
Software	**Version**
OS	Ubuntu 22.04 LTS
Kernel	Linux 6.2
fio	3.35
Cachelib	0.10.2

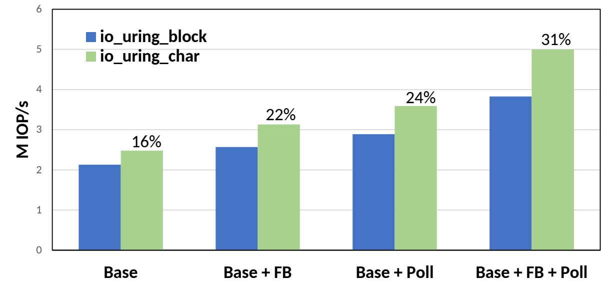
Table 3: Experimental configuration

7.1 Efficiency Characterization

The SSD used for this evaluation is notably optimized for 512b random reads and can show up to 5M IOPS as per its specification [41]. This is why we focus only

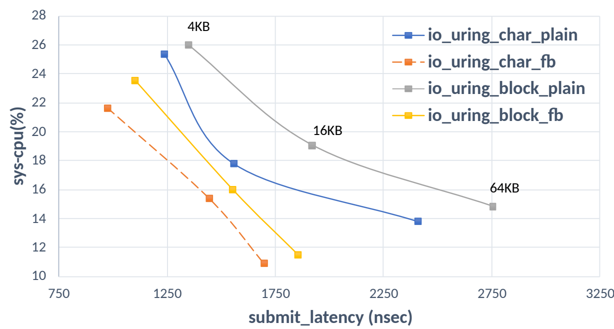


(a) Peak performance comparison on general kernel config

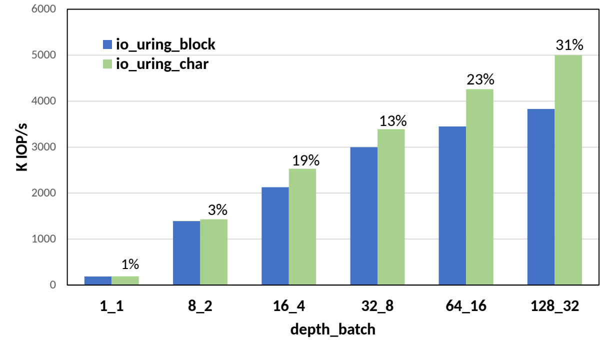


(b) Peak performance comparison on optimized kernel config

Figure 8: io_uring_char vs io_uring_block peak performance comparison



(a) Fixed-buffers effect on cpu-util and slat with increasing block size



(b) Scalability across queue-depths on optimized kernel config

Figure 9: io_uring_char vs io_uring_block scalability across queue-depths and fixed-buffers effect

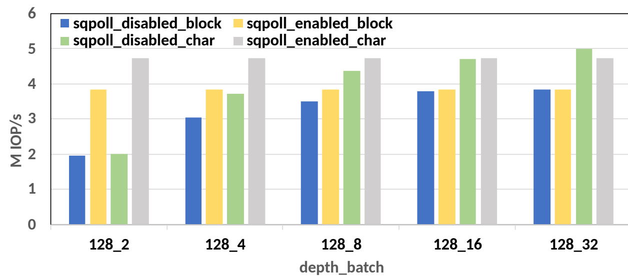


Figure 10: SQPoll and batching effect

on read-only workload, as this helps to reveal the software overheads and impact of optimizations readily. We use two kernel configurations to refine the test setup for overhead/efficiency measurements. The first one is the default configuration. The second one is a more performance-friendly configuration with CONFIG_RETPOLINE and CONFIG_PAGE_TABLE_ISOLATION options disabled. The kernel added these options to mitigate the Spectre [46] and Meltdown [50] hardware vulnerabilities. However, these come at the cost of performance overhead [9, 56].

Peak performance using single CPU core: We saturate the SSD to its maximum read performance, i.e., 5M IOPS. To that end, we measure the individual and combined impact of two knobs that elevate efficiency - (i) FB, which refers to fixed-buffers (Section 4.2.3), and (ii) poll, which refers to completion polling (Section 4.2.4). For this test, `t/io_uring` is bounded to a single CPU core, and it issues 512b random read at queue-depth 128 with batch size set to 32. Figure 8(a) shows the result on default kernel config. Fixed-buffer shows higher IOPS as the processing overhead of mapping buffers is minimized. Poll also shows improved numbers as interrupt-processing and context-switching overhead goes away. The performance of the io_uring passthrough path is better than the io_uring block path in all four cases. When both the knobs (FB and Poll) are combined, performance reaches its peak. Block I/O reaches up to 2.9M IOPS, while passthrough red I/O goes 35% higher and reaches 3.9M IOPS. However, SSD is capable of higher IOPS. Therefore, we repeat the test with an optimized kernel config. Figure 8(b) shows the results. There is notable improvement across all metrics. Block I/O elevates to 3.83M IOPS, while passthrough I/O goes 31% higher and saturates the SSD at 5M IOPS.

The reason is that I/O submission via the io_uring passthrough path involves less processing than the io_uring block path. It skips the attempts to split, merge, and i/o

scheduling. Table 4 shows the execution time of the respective io_uring handlers in the kernel (optimized config). The time these functions take corresponds to the time taken to submit the request to the device. The block-path handler, `io_read`, takes 209 nanoseconds for a single submission. The passthrough handler is 31% leaner and takes 144 nanoseconds for the submission.

Handler (io_uring)	Execution Time (nsec)
`io_read` (block)	209
`io_uring_cmd` (passthrough)	144

Table 4: Profiling of submission path

Scalability across queue-depths: We use `t/io_uring` to issue 512b random reads and vary the queue-depths (ranging from 1 to 128) and batch sizes (ranging from 1 to 32). Figure 9(b) shows the IOPS comparison between block and passthrough paths. At single queue-depth, utilization of the device bandwidth is lowest, and both paths yield the same performance. This is expected and denotes the synchronous I/O performance. As the queue-depth amplifies, parallel-processing capabilities of software and hardware get leveraged better, exhibiting a consistent increase in IOPS. A leaner submission path matters more when I/O requests arrive at a higher rate. At queue-depth 16, passthrough can process 19% more requests, which goes up to 31% at 128 queue-depth. **Cpu utilization and submission-latency**: comparison when fixed-buffer is enabled for block and passthrough I/O path. For this test, we use fio random read workload with single queue-depth and varying block sizes - 4 KB, 16 KB, and 64 KB. Figure 9(a) shows the result. In general, submission latency increases with the larger record size. This is because during the submission, (i) physical pages (usually 4KB in size) backing the I/O buffer need to be locked, and (ii) DMA (direct memory address) mapping for these pages needs to be done. A larger I/O buffer involves more physical pages, so it takes more time to perform the aforementioned steps. With a smaller block size, the submission and completion rate is high. But as we shift to large record sizes, the workload becomes more I/O bound. Therefore, CPU utilization is higher for 4KB record size. Fixed-buffer variants (of block and passthrough path) exhibit reduced submission latency and CPU cost of the I/O. io_uring char with fixed-buffer produces the most optimal combination of submission latency and CPU utilization.

SQPoll and batching: We use `t/io_uring` to issue 512b random reads with queue-depth set to 128 and vary batch sizes (ranging from 2 to 32). To reduce the contention and variance across multiple runs, we affine the sqpoll thread on a CPU core, which differs from the core to which t/io_uring is bounded. This is achieved by specifying the `IORING_SETUP_SQ_AFF` flag during io_uring's ring setup phase. Figure 10 compares the block and passthrough path with the SQPoll option disabled/enabled. SQPoll helps eliminate system call costs. With lesser batching (which would lead to more syscalls), enabling SQPoll results in better performance for both block and passthrough paths. With a batch size of 2, we get a 136% better performance by enabling the SQPoll option for io_uring passthrough. Both batching and SQPoll provide a means to reduce the syscall cost, but SQPoll requires an extra CPU core so that its active polling loop does not collide with the application thread that needs to submit the I/O.

7.2 Data-placement in Cachelib

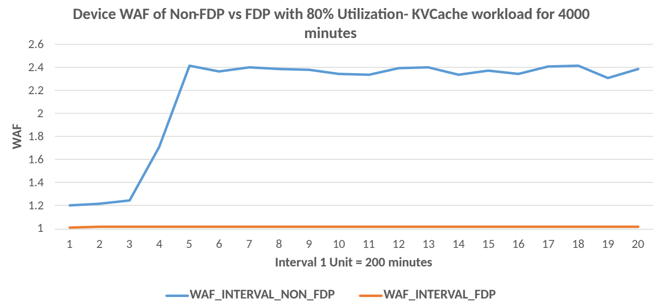


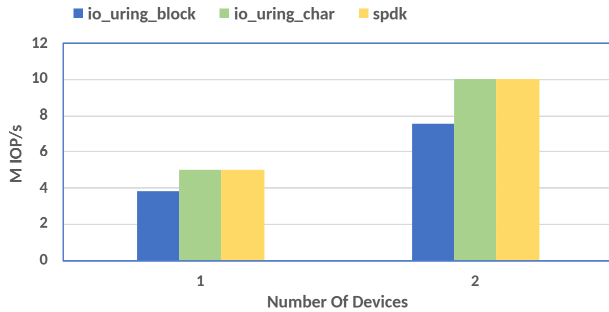
Figure 11: Cachelib WAF comparison: with and without FDP

Cachelib internally uses two I/O engines for data handling in SSD storage: **BigHash** and **BlockCache**. BigHash handles data of small sizes and random writes in 4K sizes. The large item engine, BlockCache, issues a sequential flash-friendly workload for data management. The leading cause of high WAF is the intermixing of these two I/O patterns in the physical NAND media and the resultant impact on SSD garbage collection. NVMe FDP commands allow the Host to send write hints to the SSD to avoid intermixing within the physical media. We modified Cachelib to use the I/O Passthru interface to send different placement identifiers with BigHash and BlockCache writes. The changes are being discussed for inclusion in the Cachelib upstream repository.

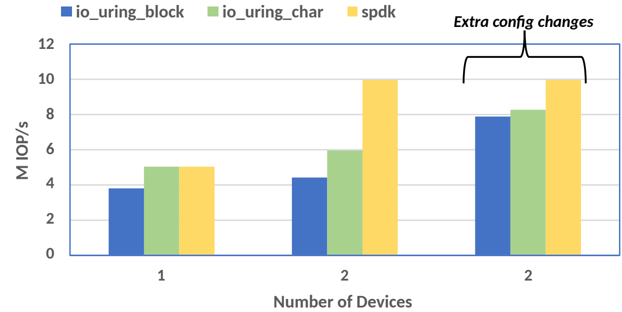
The evaluation was done using the built-in Cachebench tool. Cachebench can replay the Meta production workloads available from the Cachelib website [4]. We have used the write-only KVCache production workload for the experiments. We ran the workload for about 66 hours, and the resulting WAF comparison is shown in Figure 11. Without the placement hints, the intermixing occurs, and SSD WAF soars above 2. However, with the placement hints, intermixing reduces, and WAF remains close to 1.

7.3 Comparison against SPDK

We extend the peak performance test on two drives and compare scalability among io_uring_block, io_uring_char, and SPDK paths. We use the SPDK `perf` tool, which has minimal overhead during benchmarking. We used a distinct CPU core for each device in the first experiment. So, two cores are used for two devices. Figure 12(a) shows the comparison. The



(a) Peak performance with multiple devices



(b) Single core peak performance with multiple devices

Figure 12: io_uring_char vs io_uring_block vs spdk performance comparison

block path shows 7.55M IOPS with two devices, whereas the passthrough and SPDK paths show 10M IOPS. The second experiment examines the per-core scalability by forcing a single CPU core for both devices. Figure 12(b) shows the comparison. SPDK continues to saturate both the devices. Block path shows 4.4M, while passthrough reaches 6M IOPS.

Overall, I/O Passthru reduces the per-core efficiency gap but is still far from kernel-bypass solutions like SPDK. There are multiple reasons:

- The SPDK application (`perf` in this case) gets single-user luxury due to exclusive ownership of the NVMe device. It does not involve any extra code that must be written to ensure sharing and synchronization among multiple device users.
- I/O Passthru needs to use the block layer for its features (Section 4.4), such as hardware-queue abstraction, tag management, timeout/abort support, etc. The features come at the expense of extra processing in the I/O path.
- A few features do not fit the passthrough path, e.g., writeback-throttling and Block-cgroups. Turning off these features (by altering kernel config) cuts the processing and improves the I/O performance. The forthcoming 6.8 kernel skips these for passthrough I/O and does not require config changes. Beyond these, there are more general kernel configs that affect the I/O performance nonetheless. We turn off the forced preemption [7] and set the timer frequency to 100 Hz [8]. Figure 12(b) shows that, with extra config changes, block I/O performance improves to 7.9M, and passthrough I/O improves to 8.3M. Given the numerous kernel configuration choices, we feel more performance tuning is possible than we have explored here.

8 Discussion

8.1 I/O Passthru versus File systems

Relevance against file systems. Does passthrough make sense when Linux offers many stable and mature file systems such as XFS, BTRFS, and Ext4? We see two reasons to think that it does:

First, the maturity of these file systems comes in the way of embracing the emerging hardware. Production file systems get stability after going through battle-testing for a decade or so. Therefore, this stability is prioritized over adopting novel storage interfaces. In some cases, storage interfaces either change over a short period or do not get widespread adoption. Such cases pose the risk of bloated code and put a maintenance burden on the file system maintainers. Passthrough helps to consume new storage innovation readily in user space where real-world usefulness can be established. The compelling innovations can then find their way into robust file systems and other mature parts of the kernel.

Second, some large-scale storage systems have drifted away from file systems due to a multitude of reasons involving low performance, less control, and rigidity towards new hardware. Ceph [59] moved away from file systems and developed a new storage backend, BlueStore, which stores data directly on the raw storage device. BlueStore is the default storage backend, and it has been reported that 70% of Ceph users use this in production [26]. Aerospike also uses SSD directly using Linux direct I/O [2]. SPDK-based solutions do away with file systems. I/O Passthru presents a new choice to design storage backends with higher control, performance, and agility to embrace new hardware.

Performance against file systems. Kernel file systems create extra functionality above the block device, so their performance is usually capped by what is possible for block I/O. But FS-driven buffered I/O can perform better than block/passthrough I/O when it completes from DRAM (page cache) without causing thrashing. Table 5 compares filesystem buffered I/O performance with passthrough I/O. We use two types of fio random-read workloads, which vary in size - 8G and 32G. Both workloads spawn eight jobs, each doing 1GB I/O in the first case and 4GB I/O in the second case. FS buffered I/O performs better when the I/O size is less than the DRAM size (i.e., 16G) but worse when the workload cannot fit in the DRAM.

Read K-IOPS	I/O Size	
	8GB	32GB
Ext4 (buffered)	5767	2046
XFS (buffered)	5817	1915
Passthrough	4536	4524

Table 5: Randread performance comparison

8.2 Multi-tenancy and SQ/CQ limits

I/O Passthru does not involve dedicating the resources to a single application. Each io_uring ring (SQ/CQ pair) is a piece of preallocated memory that the application gets. This allocation is subject to the per-process limits. The application can use the same ring to do I/O on multiple files, as each SQE takes a distinct file handle as input. As for NVMe SQ/CQ, the upper limit comes from the NVMe SSD. The block layer abstracts available NVMe SQ/CQ under the per-core queues. The application that gets scheduled on a particular core submits its I/O to the underlying NVMe SQ mapped to that core. The architecture ensures that multiple applications run concurrently without reserving the hardware resources.

9 Related Work

SPDK allows applications to skip the abstraction layers and work directly with NVMe devices. The application needs to link with SPDK NVMe-driver to make use of it. However, SPDK NVMe-driver is a user-space library that maps the entire PCI bar to a single application. SPDK users face challenges when having to support multi-tenant deployment. The SPDK NVMe driver can operate only in polled mode. Also, storage is highly virtualized in a cloud environment, and root/admin access to raw PCIe devices is not feasible.

The abbreviation "ng" for the NVMe generic interface is inspired by "sg," which represents the SCSI generic interface. The sg driver, part of the Linux kernel SCSI subsystem, creates the SCSI generic interface [10]. The sg driver allows user applications to send SCSI commands to the underlying SCSI device. This communication from the user-space is done on character device node `/dev/sgX`, with syscalls such as write, read, and ioctl. Synchronous communication is done via `SG_IO` ioctl, which is analogous to `NVME_IOCTL_IO64_CMD` ioctl provided by NVMe(Section 4.2.2). Asynchronous communication using the sg interface is unhandy as it does not interface with io_uring or Linux AIO [13]. Instead, this requires pairing two system calls (read followed by a write) and signal handling [21,22]. `io_uring command` opens up an excellent way to upgrade the async communication mechanism of the SCSI generic interface.

Netlink sockets allow exchanging information in an async fashion between kernel and user-space [53, 58]. However, the netlink interface is designed for networking use cases and not for generic file I/O [34]. Some prior works proposed asynchronous ioctl via io_uring.

Pavel [33] and Hao [60] implemented by calling synchronous VFS ioctl handler in the io_uring worker context. This was anything but efficient (as Figure 3 shows). Kanchan et al. [42] early approach was tied to block-device and had allocation overhead. Jens [28] proposed a more generic and efficient approach involving `SQE` overlay. However, the `SQE` overlay did not forge ahead as (i) it provided 40 bytes of free space, which was insufficient for NVMe passthrough commands, and (ii) it brought certain plumbing unpleasantness in io_uring code. These were overcome after the introduction of `Big SQE` and cemented the proposal described in this paper.

10 Conclusion

Many new storage features/interfaces do not fit well within the block layer and face adoption changes due to the absence of appropriate syscall interfaces in Linux. Consequently, early adopters are left with two options: (i) use synchronous NVMe passthrough on block interface that may or may not exist, or (ii) switch to kernel-bypass solutions. We create a new alternative by adding a new passthrough path in the kernel. This path combines an always-available NVMe character interface with io_uring. Overall, this opens up an efficient way to use any current/future NVMe feature with the mainline kernel itself, i.e., all NVMe features with zero code in the kernel. We integrate this path to various user-space libraries/tools and present examples of how this can ease the enablement of FDP SSD, End-to-end data protection, and computational storage. As for efficiency, results demonstrate that the new passthrough path outperforms the existing block I/O path.

We also introduce an alternative of `ioctl` within io_uring. The `io_uring_command` infrastructure ensures that io_uring capabilities are not limited to existing mechanisms (i.e., classical read/write or other established syscalls) but will also be available to apply on new primitives. As is the case between host-system and storage, there will always be a requirement to communicate between user-space and kernel in a way that has not been imagined before. New pathways will remain in need. We hope `io_uring_command` will significantly ease up building efficient pathways between user-space and kernel.

Acknowledgement

We would like to thank our shepherd Sungjin Lee and the anonymous reviewers for their valuable feedback. We extend our appreciation to Joel Granados, Minwoo Im, Stefan Roesch, Vincent Kang Fu, Luis Chamberlain, and Christoph Hellwig for their contributions to I/O Passthru.

References

[1] Adding new system call. https://docs.kernel.org/process/adding-syscalls.html.

[2] Aerospike using raw storage. https://docs.aerospike.com/server/operations/plan/ssd/ssd_setup.

[3] Cachelib. https://github.com/facebook/CacheLib.

[4] Evaluating ssd hardware for facebook workloads. https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_FB_HW_eval/.

[5] ioctl, man page. https://man7.org/linux/man-pages/man2/ioctl.2.html.

[6] io_uring_enter, man page. https://man7.org/linux/man-pages/man2/io_uring_enter.2.html.

[7] Kernel config for preemption. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/kernel/Kconfig.preempt.

[8] Kernel config for timer frequency. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/kernel/Kconfig.hz.

[9] Kernel pti and its overhead. https://www.kernel.org/doc/html/latest/x86/pti.html.

[10] Kernel. scsi generic. https://docs.kernel.org/scsi/scsi-generic.html.

[11] Kioxia fl6 ssd. https://www.kioxia.com/en-jp/business/ssd/enterprise-ssd/fl6.html.

[12] Linux 5.6 i/o scheduler benchmarks. https://www.phoronix.com/review/linux-56-nvme.

[13] Linux aio. https://github.com/littledan/linux-aio.

[14] Linux kernel repository. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git.

[15] Lsm - selinux. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/admin-guide/LSM/SELinux.rst.

[16] Lsm - smack. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/admin-guide/LSM/Smack.rst.

[17] Nvme 2.0 - tp 4146, fdp. https://nvmexpress.org/wp-content/uploads/NVM-Express-2.0-Ratified-TPs_12152022.zip.

[18] nvme-cli utility. https://github.com/linux-nvme/nvme-cli.

[19] Nvme command set. https://nvmexpress.org/developers/nvme-command-set-specifications.

[20] Nvme data protection. https://www.ripublication.com/ijaer19/ijaerv14n7_10.pdf.

[21] Scsi generic, async usage. https://tldp.org/HOWTO/SCSI-Generic-HOWTO/async.html.

[22] Scsi generic, theory of operation. https://tldp.org/HOWTO/SCSI-Generic-HOWTO/theory.html.

[23] Snia. computational storage. https://www.snia.org/computational.

[24] spdk-release. https://github.com/spdk/spdk/releases/tag/v22.09.

[25] Z-ssd. https://semiconductor.samsung.com/ssd/z-ssd/.

[26] AGHAYEV, A., WEIL, S., KUCHNIK, M., NELSON, M., GANGER, G. R., AND AMVROSIADIS, G. File systems unfit as distributed storage backends: lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), pp. 353–369.

[27] ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND PRABHAKARAN, V. Removing the costs of indirection in flash-based {SSDs} with nameless writes. In *2nd Workshop on Hot Topics in Storage and File Systems (HotStorage 10)* (2010).

[28] AXBOE, J. async ioctl. https://lwn.net/Articles/844875/.

[29] AXBOE, J. io_uring doc. https://kernel.dk/io_uring.pdf.

[30] AXBOE, J., ET AL. Flexible i/o tester. https://github.com/axboe/fio.

[31] AXBOE, J., ET AL. Liburing. https://github.com/axboe/liburing.

[32] AXBOE, J., ET AL. t/io_uring utility. https://github.com/axboe/fio/blob/master/t/io_uring.c.

[33] BEGUNKOV, P. async ioctl. https://lore.kernel.org/all/f77ac379ddb6a67c3ac6a9dc54430142ead07c6f.1576336565.git.asml.silence@gmail.com/.

[34] BERGMANN, A. How to not invent kernel interfaces. In *LinuxConf Europe 2007 Conference and Tutorials, 2.-5. září 2007* (2007).

[35] BJØRLING, M. Zone append: A new way of writing to zoned storage. *Santa Clara, CA, February. USENIX Association.[Cited on page.]* (2020).

[36] BJØRLING, M., AXBOE, J., NELLANS, D., AND BONNET, P. Linux block io: Introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference* (New York, NY, USA, 2013), SYSTOR '13, Association for Computing Machinery.

[37] 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.

[38] GONZÁLEZ, J. Zoned namespaces: Standardization and linux ecosystem. *SDC EMEA* (2020).

[39] HELLWIG, C. Remove write-hint. https://lore.kernel.org/all/20220304175556.407719-2-hch@lst.de/.

[40] IM, M. generic per-namespace chardev. https://lore.kernel.org/linux-nvme/20210421074504.57750-2-minwoo.im.dev@gmail.com/.

[41] INTEL. Optane p5800x spec. https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/data-center-ssds/optane-ssd-p5800x-p5801x-brief.html.

[42] JOSHI, K. async ioctl. https://lore.kernel.org/linux-nvme/20210127150029.13766-1-joshi.k@samsung.com/.

[43] JOSHI, K., AND S, S. Towards copy-offload in linux nvme. *SDC* (2021).

[44] JUNG, T., LEE, Y., AND SHIN, I. Openssd platform simulator to reduce ssd firmware test time. *Life Science Journal 11*, 7 (2014).

[45] KNIGHT, F. Storage data movement offload. *NetApp, Sep* (2011).

[46] KOCHER, P., HORN, J., FOGH, A., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., ET AL. Spectre attacks: Exploiting speculative execution. *Communications of the ACM 63*, 7 (2020), 93–101.

[47] KWAK, J., LEE, S., PARK, K., JEONG, J., AND SONG, Y. H. Cosmos+ openssd: Rapid prototype for flash storage systems. *ACM Trans. Storage 16*, 3 (jul 2020).

[48] LEI, M. ublk io_uring_cmd. https://lore.kernel.org/linux-block/20220628160807.148853-2-ming.lei@redhat.com/.

[49] LEITAO, B. uring_cmd network-socket. https://lore.kernel.org/lkml/20230627134424.2784797-1-leitao@debian.org/.

[50] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., ET AL. Meltdown: Reading kernel memory from user space. *Communications of the ACM 63*, 6 (2020), 46–56.

[51] LUND, S. A., BONNET, P., JENSEN, K. B., AND GONZALEZ, J. I/o interface independence with xnvme. In *Proceedings of the 15th ACM International Conference on Systems and Storage* (2022), pp. 108–119.

[52] LWN. The rapid growth of io_uring. *https://lwn.net/Articles/810414/*.

[53] NEIRA-AYUSO, P., GASCA, R. M., AND LEFEVRE, L. Communicating between the kernel and user-space in linux using netlink sockets. *Software: Practice and Experience 40*, 9 (2010), 797–810.

[54] PETERSEN, M. K. Copy offload. here be dragons. *https://oss.oracle.com/~mkp/docs/xcopy.pdf*.

[55] PETERSEN, M. K. Dif, dix and linux data integrity. *Oracle, downloaded* (2010), 25.

[56] PROUT, A., ARCAND, W., BESTOR, D., BERGERON, B., BYUN, C., GADEPALLY, V., HOULE, M., HUBBELL, M., JONES, M., KLEIN, A., ET AL. Measuring the impact of spectre and meltdown. In *2018 IEEE High Performance extreme Computing Conference (HPEC)* (2018), IEEE, pp. 1–5.

[57] REN, Z., AND TRIVEDI, A. Performance characterization of modern storage stacks: Posix i/o, libaio, spdk, and io_uring. In *Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems* (2023), pp. 35–45.

[58] ROSEN, R. Netlink sockets. In *Linux Kernel Networking*. Springer, 2014, pp. 13–35.

[59] 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.

[60] XU, H. async ioctl. *https://lore.kernel.org/all/1604303041-184595-1-git-send-email-haoxu@linux.alibaba.com/*.

[61] ZHANG, J., LI, P., LIU, B., MARBACH, T. G., LIU, X., AND WANG, G. Performance analysis of 3d xpoint ssds in virtualized and non-virtualized environments. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)* (2018), IEEE, pp. 1–10.

A Artifact Appendix

Abstract

The evaluated artifact is provided in a git repository and contains the scripts used for running the experiments presented in this paper.

Scope

The artifact contains the scripts to reproduce the results obtained in Figure 8, Figure 9, Figure 10 and Figure 11.

Contents

The artifact contains the steps to build and install linux, links to patches for kernel and userspace contributions. It also contains the scripts used for performance benchmarks and cachelib experiments in the benchmark and cachelib-experiments subdirectory respectively. Also, each subdirectory has a separate README file, specifying the usage instructions.

Hosting

The artifact is available at https://github.com/anuj7781/io-passthru. All necessary instructions are provided in the README.md file. We encourage the users to use the latest version of the repository, since it may include bug fixes.

Requirements

The experiments can be run on any Linux machine (with 6.2 kernel). The benchmark experiments can be run on any NVMe drive, while the cachelib experiments can be run only on a FDP device. In order to reproduce the results, one needs to use the setup mentioned in Table 3.

Notes

Metis: File System Model Checking via Versatile Input and State Exploration

Yifei Liu, Manish Adkar, Gerard Holzmann*, Geoff Kuenning+, Pei Liu,
Scott A. Smolka, Wei Su, and Erez Zadok
*Stony Brook University, *Nimble Research, +Harvey Mudd College*

Abstract

We present *Metis*, a model-checking framework designed for versatile, thorough, yet configurable file system testing in the form of input and state exploration. It uses a nondeterministic loop and a weighting scheme to decide which system calls and their arguments to execute. Metis features a new *abstract state* representation for file-system states in support of efficient and effective state exploration. While exploring states, it compares the behavior of a file system under test against a reference file system and reports any discrepancies; it also provides support to investigate and reproduce any that are found. We also developed RefFS, a small, fast file system that serves as a reference, with special features designed to accelerate model checking and enhance bug reproducibility. Experimental results show that Metis can flexibly generate test inputs; also the rate at which it explores file-system states scales nearly linearly across multiple nodes. RefFS explores states 3–28× faster than other, more mature file systems. Metis aided the development of RefFS, reporting 11 bugs that we subsequently fixed. Metis further identified 12 bugs from five other file systems, five of which were confirmed and with one fixed and integrated into Linux.

1 Introduction

File system testing is an essential technique for finding bugs [43] and enhancing overall system reliability [27], as file-system bugs can have severe consequences [53, 92]. Effective testing of file systems is challenging, however, due to their inherent complexity [4], including many corner cases [87], myriad functionalities [8], and consistency requirements (*e.g.*, crash consistency [64, 72]). Developers have created various testing technologies [59, 71, 86] for file systems, but new bugs (both in-kernel and non-kernel) continue to emerge on a regular basis [42, 43, 85].

To expose a file-system bug, a testing tool must execute a particular system call using specific inputs on a given file-system state [52, 53, 87]. For example, identifying a well-known Ext4 bug [48] requires a write operation on a file initialized with a 530-byte data segment. In this case, the write operation is an input, and the file with a specific size constitutes (part of) the file-system state. Recent work [9, 52] also underscored the importance of adequately covering both file-system inputs and states during testing. While existing testing technologies seek to cover a broad range of file systems' functionality, they often do not, however, integrate coverage of *both* file-system inputs and states [12, 43, 59, 85]. For example, handwritten regression tools like xfstests [71] can achieve good test coverage of specific file-system features [4, 58], but do not comprehensively cover syscall inputs; similarly, fuzzing techniques (*e.g.*, Syzkaller [25]) are designed to maximize code—not input—coverage [40].

Both the input and state spaces of file systems are too vast to be completely explored and tested [10, 21], so it is better to leverage finite resources by focusing on the most pertinent inputs and states [52, 86, 88]. For example, metadata-altering operations, such as `link` and `rename`, and states with a complex directory structure are more frequently utilized in POSIX-compliance testing [67]. Existing testing technologies also lack the versatility to test specific inputs and states [25, 59, 71]. Thus, new testing tools and techniques are needed [52, 53] to avoid under-testing (which could miss potential bugs) or over-testing (which wastes resources that may be better deployed elsewhere).

This paper presents Metis, a novel model-checking framework that enables thorough and versatile input and state space exploration of file systems. Metis runs two file systems concurrently: a file system under test and a reference file system to compare against [26]. Metis issues file-system operations (*i.e.*, system calls with arguments) as inputs to both file systems while simultaneously monitoring and exploring the state space via graph search (*e.g.*, depth-first search [31]).

To compare the relevant aspects of file-system states, we first abstract them and then compare the abstractions. The abstract states include file data, directory structure, and essential metadata; abstract states constitute the state space to be explored. Metis first nondeterministically selects an operation and then fills in syscall arguments through a user-specified weighting scheme. Next, it executes the same operation in both file systems and then compares both systems' abstract states. Any discrepancy is flagged as a potential bug. Metis evaluates the post-operation states to decide if a state has been previously explored; if so, it backtracks to a parent state and selects a new state to explore [31]. Metis continuously tests new file-system states until no additional unexplored states remain, logging all operations and visited states for subsequent analysis. Metis's replayer can reproduce potential bugs with minimum time and effort.

Metis effectively addresses the common challenges of model checking [16, 31] file systems. It checks file-system implementations directly, eliminating the need to build a formal model [61]. To manage large file-system input and state spaces, Metis enables parallel and distributed exploration [33] across multiple cores and machines. Metis works with any kernel or user file system, and does not require any specific utilities nor any modification or instrumentation of the kernel or the file

system. It detects bugs by identifying behavioral discrepancies between two file systems without the need for oracles or external checkers, thus simplifying the process of applying Metis to new file systems. With few constraints, Metis is well suited for testing file systems that are challenging for other testing approaches, *e.g.*, file system fuzzing [43], that require kernel instrumentation and utilities. Nevertheless, the quality of the reference file system is pivotal for assessing the behavior of other file systems [26]. We therefore developed RefFS as Metis's reference file system. RefFS is an in-memory user-space POSIX file system with new APIs for efficient state checkpointing and restoration [73, 86]. Prior to using RefFS as our reference file system, we used Ext4 as the reference to check RefFS itself; Metis identified 11 RefFS bugs that we fixed during that process. Subsequently, we deployed 18 distributed Metis instances to compare RefFS and Ext4 for one month, totaling 557 compute days across all instances and executing over 3 billion file-system operations without detecting any discrepancy. This ensured that RefFS is robust enough to serve as Metis's (fast) reference file system.

Our experiments show that Metis can configure inputs more flexibly and cover more diverse inputs compared to other file-system testing tools [25, 59, 71]. Metis's exploration rate scales nearly linearly with the number of Metis instances, also known as verification tasks (VTs). Despite being a user-level file system, RefFS's states can be explored by Metis 3–28× faster than other popular in-kernel file systems (*e.g.*, Ext4, XFS, Btrfs). Using Metis and RefFS, we discovered 12 potential bugs across five file systems. Of these, 10 were confirmed as previously unknown bugs, five of which were confirmed by developers as real bugs. Moreover, one of those bugs—which the developers confirmed existed for 16 years—and the fix we provided, was recently integrated into mainline Linux.

In sum, this paper makes the following contributions:

1. We designed and implemented Metis, a model-checking framework for versatile and thorough file-system input and state-space exploration.
2. We designed and implemented an effective abstract state representation for file systems and a corresponding differential state checker.
3. We designed and implemented the RefFS reference file system with novel APIs that accelerate and simplify the model-checking process.
4. Using RefFS, we evaluated Metis's input and state coverage, scalability, and performance. Our results show that Metis, together with RefFS, not only facilitates file-system development but also effectively identifies bugs in existing file systems.

2 Background and Motivation

In this section, we first introduce the procedures and challenges for testing and model-checking file systems. We then discuss two vital dimensions for file system testing: input and state. We demonstrate the challenges of achieving versatile and comprehensive coverage of both inputs and states.

File system testing and model checking. File systems can be tested statically or dynamically. Static analysis [9, 57] evaluates the file system's code without running it; while useful, it struggles with complex execution paths that may depend on runtime state. Our work therefore emphasizes dynamic testing—executing and checking file systems in real-time scenarios [12, 59, 67]. Generally, dynamic testing involves (1) crafting test cases using system calls, (2) initializing the file system, (3) running the test cases, and (4) post-execution validation of file system properties. Hence, the quality of test cases directly affects the testing efficacy.

Model checking is a formal verification technique that seeks to determine whether a system satisfies certain properties [16, 77]. The model is typically a state machine, and the properties, usually expressed in temporal logic, are checked using state-space exploration [15]; here, each state represents a snapshot of the system under investigation. To automate this process, model checkers (*e.g.*, SPIN [31]) are used to generate the state space, verify property adherence, and provide a counterexample when a property is violated.

Extracting a model from a system implementation can be challenging, especially for large systems like file systems [86, 87]. Thus, recent work on implementation-level model checking [86, 87] seeks to check the implementation directly (without a model). Such approaches [86] require one to create new, specialized checkers to test new file systems, and these checkers are typically focused on a limited range of bugs, such as crash-consistency bugs [86, 87]. The ongoing challenge is to simplify implementation-level file-system model checking so that using it does not require extensive effort or significant expertise in model checking and file systems, while at the same time being able to identify a wide range of bugs.

Covering system calls and their inputs. We refer to the system calls (syscalls) and their arguments as *inputs* or *test inputs* because syscalls are commonly used by user-space applications—and thus testing tools—to interact with file systems [22, 81]. Thoroughly testing file system inputs is challenging. While file-system–related syscalls represent only a subset of all Linux syscalls [7, 74], each syscall has multiple arguments, and the potential value range for these arguments is vast [52, 74]. For example, `open` returns a file descriptor, accepting user-defined arguments for `flags` and `mode` in addition to `pathname`. Both `flags` and `mode` are bitmaps with 23 and 17 bits, respectively, representing many possible combinations. The bits represented in `flags` alone have 2^{23} possible values, leading to an aggregate input space of 2^{40}. Similarly, `write` and `lseek` take 64-bit-long byte-count arguments that have a large input domain of 2^{64} possible values. Nevertheless, it is vital to test as many representative syscall inputs as possible.

Fully testing all syscalls with every potential argument is impractical [25, 37]. Instead, a sensible approach [45, 52] is to *segment* a large input space into multiple, disjoint input partitions—called *input space partitioning* [39, 52, 78]. How

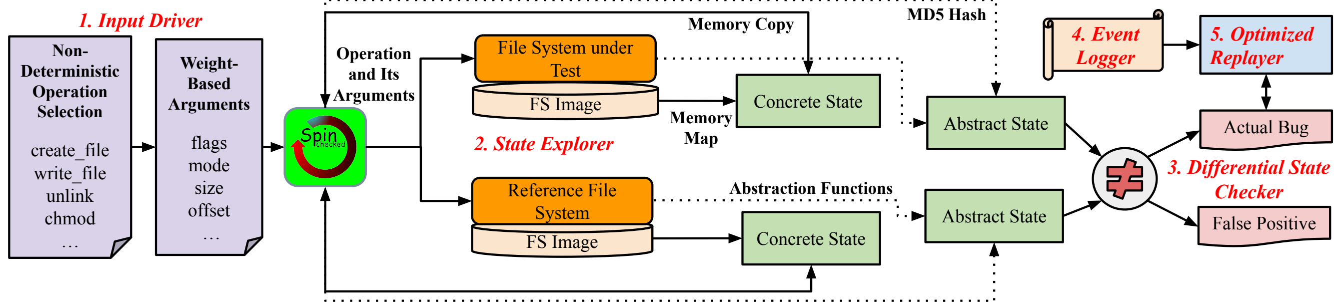


Figure 1: Metis architecture and components. From left to right, Metis generates syscalls and their arguments that are executed by both file systems, determines resulting states, and checks for discrepancies between states. The Logger records all the operations for convenient bug replay by the Replayer. The SPIN model checker stores previous state information for state exploration.

much a testing tool examines input partitions is called *input coverage* [30, 45, 75]. Utilizing input partitions and coverage, testing tools can target the coverage of different partitions—each representing a subset of analogous test inputs. Intuitively, file system developers recognize the need to, say, separately test critical I/O write sizes of 512 and 4096; conversely, once one tests an I/O size of, say, 5000 bytes, the gains from testing subsequent adjacent sizes (*e.g.*, 5001, 5002, ...) quickly diminish.

To compute input coverage, we categorized each syscall's arguments into four classes [7, 52, 74]: (i) identifiers (*e.g.*, file descriptors), (ii) bitmaps (*e.g.*, `open` flags), (iii) numeric arguments (*e.g.*, `write` size), and (iv) categorical arguments (*e.g.*, `lseek` "whence"). We partitioned the input space using type-specific methods. For example, bitmaps are partitioned by each flag and certain combinations thereof. Numeric arguments are partitioned by boundary values (*e.g.*, powers of 2 [38]). Our goal is to achieve thorough input coverage while configuring it based on test strategies to customize the overall search space. To the best of our knowledge, no existing file system testing method is specifically designed for comprehensive input coverage, nor are there any techniques to flexibly define the input's coverage.

Challenges of testing file system states. In file system testing, the *state* refers to the content, status, and full context of the file system at a given point in time [21, 73]. Comprehensive state exploration is important as certain bugs manifest exclusively under specific states [48, 53, 76]. Numerous file system states can be explored when some existing testing approaches [59, 71] execute operations. Yet the majority of these approaches lack state tracking—the ability to record and identify previously or similarly visited states—thus wasting resources [86]. The challenges are thus twofold: state definition and efficient state tracking.

Defining file system states involves a tradeoff, because components such as on-disk content, in-memory data, configuration, kernel context, and device types are all candidates for inclusion in the state [21]. An overly detailed state definition can render state exploration infeasible due to resources spent on visiting multiple states that should be treated as if they were identical [16]. Conversely, an overly narrow definition can skip key states and potentially miss defects [11]. Therefore, one should be able to define the state space flexibly, so it contains all desired file system attributes while maintaining a manageable state space.

Due to massive state spaces, state tracking incurs considerable overhead, thus slowing the entire exploration process. While model checkers provide a mechanism for state exploration [31] with state tracking and certain optimizations, they still have to contend with the state explosion problem—a significant challenge where the number of system states grows exponentially with the number of system variables, making state exploration computationally impractical [16]. In file systems, this issue is exacerbated by the inherently slow nature of I/O. An alternative approach is to partition the state-exploration process across multiple instances, with each instance exploring a certain portion of the state space; doing so requires a sophisticated design for diversified, parallel exploration [33].

3 Design

In this section, we describe Metis's design principles and operation. We explain how Metis meets the challenges of exploring file system inputs and states, and how it provides versatility.

Metis architecture. As shown in Figure 1, Metis has five main components: (1) Input Driver, (2) State Explorer, (3) Differential State Checker, (4) Event Logger, and (5) Optimized Replayer. Each component is designed to be independent, allowing for modularity and extensibility.

The Input Driver (§3.1) generates syscalls and arguments to serve as the test inputs to both file systems. Metis is built on top of the SPIN model checker [31] to combine input selection with state exploration. The State Explorer (§3.2) extracts concrete and abstract states from both file systems and interfaces with SPIN to explore new states. The Differential State Checker (§3.3) verifies that both file systems have identical behavior after each operation, by comparing their abstract states, syscall return values, and error codes. Any discrepancies are reported by the checker and treated as potential bugs. The Event Logger and the Optimized Replayer (§3.4) help analyze reported discrepancies and reproduce potential bugs more efficiently.

3.1 Input Driver

Metis's Input Driver maintains a list of operations from which the SPIN model checker can repeatedly and nondeterministically

choose what to execute, including individual syscalls (*e.g.*, unlink) as well as meta-operations comprising a (small) sequence of syscalls (*e.g.*, the write_file operation opens a file and writes to it at a specific offset). From a given file system state, multiple potential successor states may arise. Through its nondeterministic choices of operations, Metis can effectively explore many of these options, ensuring thorough state exploration. To bound the input space, each operation randomly picks a file or directory name from a predetermined set of pathnames. The Input Driver is flexible and can generate files or directories with arbitrarily deep directory structures, long pathnames, and other unexpected scenarios such as many files inside a single directory.

We focus on state-changing operations [26] (*i.e.*, not read-only ones) as the Input Driver seeks to maximize the exploration of file system states. Currently, the Input Driver supports five meta-operations (create_file, write_file, chown_file, chgrp_file, and fallocate_file), and 10 individual syscalls (truncate, unlink, mkdir, rmdir, chmod, setxattr, removexattr, rename, link, and symlink). Adding a new operation has minimal effort of about 10 LoC. Metis exercises read-only operations such as read, getxattr, and stat after each state-changing operation, when computing file system abstract states in the State Explorer (§3.2).

After selecting the operation, Metis chooses its arguments based on a series of user-specified weights that control how often various argument partitions (§2) are tested. In the Input Driver, weights represent the probabilities assigned to different input partitions, which control testing frequencies. The method of assigning weights varies based on the argument type [7, 52]. For bitmap arguments, each bit receives a probability of being set. The number of input partitions in a bitmap argument is equivalent to its individual bit count. Given the ubiquity of powers of 2 in file systems [38], numeric arguments like write size (requested byte count) have input partitions segmented by these numbers as boundary values, rounding down to the nearest boundary. For example, write sizes ranging from 1024 to 2047 bytes (2^{10} to $2^{11}-1$) are grouped in the same partition. Assigning a weight (*e.g.*, 15%) to this partition implies a 15% chance of selecting a write size between 1024 and 2047 bytes. The total weight of all write-size partitions equals 100%. We placed 0 bytes as a distinct partition (unusual but allowed under POSIX) because the smallest power of 2 is 1, which is greater than 0. Additionally, Metis can also be configured to test only boundary values (powers of 2) such as 4096 as well as near-boundary values (± 1 from the boundary, *e.g.*, 4095/4097) that are useful for testing underflow and overflow conditions.

The choice of weights depends on the user's objectives. For example, while O_SYNC is common in crash-consistency testing [59], it is used infrequently for POSIX compliance [67]. Due to disk I/O's slow speed, many tests focus on small write sizes [12]. However, testing larger sizes can uncover size-specific bugs [67, 76]. Our objective is to ensure that Metis remains versatile and to allow one to adjust the input weights in line with the test focus.

3.2 State Exploration and Tracking

State explorer. The objective of Metis's State Explorer is to use graph traversal to conduct thorough and effective "state graph exploration," where the nodes correspond to file-system states and the edges represent transitions caused by operations [15]. Metis supports depth-first search (DFS) as the main search algorithm.

The State Explorer relies on the SPIN model checker [31] to conduct the state-space exploration. SPIN supports the Promela model-description language, and allows embedding C code in Promela code. This capability allows us to seamlessly issue low-level file-system syscalls and invoke utilities. SPIN's role is to provide optimized state-exploration algorithms (*e.g.*, DFS) and data structures to track and store the status of the state graph; thus, we do not have to implement these features in the State Explorer.

In model checking, there are two types of states: *concrete* and *abstract*. Concrete states contain all the information that describes the states of the file system being checked. Abstract states serve as signatures to identify different system states of interest during the exploration.

After each operation, the State Explorer calls the abstraction function to extract abstract states as hash values from both file systems. Every time an abstract state is created, SPIN checks whether it has already been visited by looking up the abstract state in SPIN's hash table and decides on the next action, either backtracking to a previous concrete state or continuing from the current one. Meanwhile, the State Explorer mmaps the full file-system image into memory to be tracked by SPIN as a concrete state. Concrete states are stored in SPIN's stack to allow the State Explorer to restore the full file-system state as required. To improve the performance of state exploration, we use RAM disks as backend devices for on-disk file systems. In Metis, we create both file systems with the minimum device sizes to reduce the memory consumption of maintaining concrete states and to make it easier to trigger corner cases such as ENOSPC.

File system abstract states. A *concrete state* is a reflection or snapshot of the entire (and highly detailed) file-system image, which renders it inappropriate for distinguishing a previously visited state [11]. This is because any small change to the file-system image leads to a new concrete state, even though there may be no "logical" change in the file system. For example, Ext4 updates timestamps in the superblock during each mutating operation, even if no actual change to a user-visible file was made. This substantially expands the state space, with many states differing only by minor timestamp changes, and leads to wasted resources on logically identical states. Additionally, because file systems are designed with different physical on-disk layouts, we cannot use concrete states to compare their behaviors. Therefore, we need a different state representation that includes only the essential and comparable attributes common to both file systems.

To address this problem, we defined an *abstraction function* to calculate file-system *abstract states* to distinguish unique states, and to compare file system behaviors. The abstract state contains pathnames, data, directory structure, and important metadata for

Problem	Cause of discrepancies	Solution
Different directory size for same contents	Size calculation methods	Ignore directory sizes
Different orders of directory entries	Internal data structures	Sort the output of `getdents`
FS-specific special files and directories	Internal implementations	Create an exception list of special entries
Different usable data capacities	Space reservation and utilization	Equalize free space among file systems

Table 1: Examples of false positives identified and addressed by Metis.

all files and directories (*e.g.*, mode, size, nlink, UID, and GID); we exclude any noisy attributes such as `atime` timestamps. We then hash this information to compact the abstract state for a more effective comparison. Metis supports several hash functions to compute abstract states; we evaluated the speed and collision resistance of each hash function (results elided for brevity) and chose MD5 by default as it had the best tradeoff of those characteristics.

The abstraction function deterministically aggregates key file system data and metadata, enabling comparison across different file systems. Specifically, the abstraction function begins by enumerating all files and directories in the file system by traversing it from the mount point. Their pathnames are sorted into a consistent, comparable order. We then `read` each file's contents and call `stat` to extract its important metadata mentioned above, following the pathname order. Finally, we compute the (MD5) hash based on the files' content, directory structure, important metadata, and pathnames to acquire the abstract state. Using abstract states not only prevents visiting duplicate states but also significantly reduces the amount of memory needed to track previously-visited states, owing to our lightweight hash representation, which in turn boosts Metis's exploration speed.

Tracking full file system states. In addition to abstract states, another complexity in tracking file system states is saving and restoring the concrete states when Metis needs to backtrack to a previous state (*i.e.*, when reaching an already visited state); this involves State Save/Restore (SS/R) operations for concrete states. Concrete states must contain all file system information including persistent (on-disk) and dynamic (in-memory) states. Metis can feasibly save and restore on-disk states by copying the on-disk device and subsequently copying it back. Kernel file systems (*e.g.*, Ext4 [55]) maintain states in kernel space, which is inaccessible to Metis, a user process. Similarly, user-space file systems built on libFUSE (*e.g.*, fuse-ext2 [2]) are separate processes with separate address spaces, so again Metis cannot directly track their internal state. Tracking only persistent on-disk state leads to cache incoherency, because cached in-kernel information is inconsistent with the on-disk content.

We tried and evaluated several approaches to tracking full file system states (performance results elided for brevity) including `fsync` syscall, `sync` mount option, process snapshotting [17,84], VM snapshotting [44, 46], and LightVM [54]. None of these approaches were effective due to their functional deficiencies or inefficient performance. For those reasons, we adopted the approach presented in [73] to unmount and remount the file system between *each* operation in Metis. An unmount is the *only* way to fully guarantee that no state remains in kernel memory. Remounting guarantees loading the latest on-disk state, ensuring cache coherency between each state exploration. This unmount-remount method was a compromise that ensures data coherency yet provides reasonable performance (§5.2), especially coupled with our specialized RefFS (§4).

3.3 Differential State Checker

Metis checker goals and approaches. Using only the Input Driver and State Explorer would constrain the detection of bugs to those manifesting as visible symptoms [12], such as kernel crashes. We thus needed a dedicated checker to identify cases where file systems fail silently [43] (*e.g.*, data corruption). Moreover, existing checkers usually require considerable effort to be applied to newly developed or constantly-evolving file systems. For example, since many checkers are hand-written (*e.g.*, xfstests), the testing of new file systems involves redesigning and refactoring test cases. Some checkers depend on an exact (*e.g.*, POSIX) specification or an oracle for bug detection [59, 67]: they are difficult to adapt to continuously-evolving file systems.

File systems vary considerably in terms of their developmental stages [53, 90]: mature file systems are typically more stable than new, emerging, or less popular ones [53]. Yet many still share common (POSIX) features and data-integrity requirements. Therefore, we rely on a *differential testing* approach [56], to check emerging file systems for silent bugs, eliminating the need for a detailed specification or an oracle.

We developed Metis's Differential State Checker to identify a broad range of file system bugs and facilitate file system development. Our checker can easily adapt to test new file systems; it requires no modification to the checker, only a replacement of the file system under test. Metis uses a well-tested, reliable file system as the reference file system and a less-tested, emerging one as the file system under test. After each file system operation, the Differential State Checker compares the resulting states of both file systems to detect any discrepancies. To prevent false positives, it only compares the common attributes of file systems, including their abstract states, return values, and error codes.

Eliminating false positives. As any discrepancy is reported as a potential bug, when developing Metis we found that it sometimes identified discrepancies that were not bugs (*i.e.*, false positives). We implemented measures to avoid these false positives. Table 1 summarizes several such cases including their problems, causes, and solutions.

All these discrepancies arose due to different file system designs and implementations. For instance, Ext4 has a special

lost+found directory and computes directory sizes by a multiple of the block size. In contrast, other file systems report sizes by the number of active entries and do not have a lost+found directory. Despite the same device sizes for different file systems, the available space varies due to different utilized and reserved space (*e.g.*, for metadata). To address this, we equalize free space among file systems by creating dummy files based on the differences in their available spaces.

While developing Metis, we analyzed every discrepancy we encountered and addressed all false positives. Whenever a false positive was identified, we updated the state abstraction function or file system initialization code to eliminate such instances, an infrequent process that was conducted manually. None of these solutions introduce false negatives, because they all deal with non-standardized behavior. For example, an application should not expect sorted output from getdents. Nevertheless, if a change introduces any misbehavior, Metis's Differential State Checker will report and handle it.

3.4 Logging and Bug Replay

When detecting a discrepancy, it is important to be able to analyze the operations executed by the file systems to identify and reproduce the potential bug. Thus, Metis's Event Logger records details of all file-system operations and outcomes, comprising every syscall and their arguments, return values, error codes, SS/R operations, and resultant abstract state. Additionally, the Event Logger logs file-system information such as the directory structure and important metadata to pinpoint the deviant behavior as soon as a discrepancy is detected. To reduce disk I/O, we store the runtime logs in an in-memory queue and periodically commit them to disk. Leveraging the Event Logger, we can reproduce the precise sequence of operations leading to a discrepancy found by Metis.

Metis can replay identified bugs by re-executing the operations from the start of Metis's run. This process can be time-consuming, however, if the discrepancy was detected after executing many operations and passing through numerous states [3]. So we needed a way to reproduce a discrepancy quickly. Existing test-case minimization techniques [43, 91] remove one operation from a sequence until the remaining operations can reproduce the bug; but this trial-and-error process is slow due to the abundance of I/O operations.

To replay bugs efficiently, the Optimized Replayer reproduces them using only a few operations (recorded in logs) and one (concrete state) file system image. Using SPIN, we retain concrete states in a stack, thereby capturing all file-system images along the current exploration path and allowing for bug reproduction from any desired location in the stack. Recent findings [43, 59] indicate that most bugs can be reproduced on a newly created file system using a sequence of eight or fewer operations. Accordingly, Metis uses an in-memory circular buffer to retain pointers to a few of the most recent file-system images (defaults to 10, but configurable) for quick post-bug processing. In practice, we first attempt to reproduce the bug using the most recent image (immediately preceding the bug state) along with the latest operation. If unsuccessful, we turn to the previous image and the two last operations, and so on in a similar pattern. This eliminates the need for Metis to replay the entire operation sequence from the beginning.

3.5 Distributed State Exploration

Along with performing state abstraction and setting limits on the number of files and directories, we also restrict the search depth to control the exponential growth of the state space. We set the maximum search depth to 10,000 by default [31]. If the search hits the $10{,}000^{th}$ level, Metis reverts to the prior state rather than exploring deeper. Thus, the state space becomes bounded, allowing Metis to perform an exhaustive search. Still, even with this depth restriction, the state space remains large because of the variety in test inputs and file system properties [21]. Exploring this space using a single Metis process (called a *verification task*, or VT) requires significant time.

To parallelize the state-space exploration [32] we use Swarm verification [33], which generates parallel VTs based on the number of CPU cores. Each VT examines a specific portion of the state space. To prevent different VTs from re-exploring the same states, and to avoid having to coordinate states across VTs, SPIN employs several *diversification* techniques [33], where every VT receives a unique combination of bit-state hash polynomials, number of hash functions, random-number seeds, search orders (*e.g.*, forward or in reverse) and search algorithms (*e.g.*, DFS), ensuring varied exploration paths.

We enabled these parallel and distributed exploration capabilities for Metis. The setup uses a configuration file to determine the machine and CPU core count; Metis then produces the exact VT count based on the configuration file. When Metis runs on distributed machines, each runs a handful of VTs, one per CPU core. Each VT is automatically configured with a distinct combination of diversification parameters, guiding them to explore different state space areas. Utilizing multiple Metis VTs across multiple cores and machines increases the overall speed of state exploration while testing more inputs. Every Metis VT operates independently, with its own device, mount point, and logs, without interference with other VTs. Given that VTs explore states autonomously without inter-VT communication, there is a risk of resource wastage if several VTs examine the same state [33]. We deployed multiple VTs on several multi-core machines and evaluated Metis extensively under Swarm verification (§5.2).

3.6 Implementation Details

Metis uses SPIN to achieve basic model-checking functions. The Promela modeling language [31] serves as the main interface with SPIN. We wrote 413 lines of Promela, consisting of do...od loops that repeatedly select one of a number of cases in a nondeterministic fashion. Each case issues file-system operations, performs differential checks, and records logs. The main part of Metis comprises 7,911 lines of C/C++ code that implement Metis's components and its communication with SPIN. We also created 1,230 lines of Python/Bash scripts to manage different Metis VTs and runtime setup, such as invoking

mkfs, and creating mount points and devices. We created RAM block devices as backend storage for on-disk file systems. Linux's RAM block device driver (brd) requires all RAM disks to be the same size. We modified it (renamed brd2), to allow different-sized disks for file systems with different minimum-size requirements. We used brd2 to create devices for on-disk file systems during the evaluation.

We changed 72 lines of SPIN's code (Aug 2020 version) to add dedicated hook functions for file system SS/R operations. Lastly, we added 31 lines of code to the original Swarm verification tool (Mar 2019 version) to enable more flexible compilation options and smoother compatibility with Metis.

In our experience, adding a new file system operation to Metis is straightforward. It requires only one additional case in the Promela code, amounting to about 10 lines. Most functionality in Metis is file-system-agnostic, *e.g.*, deploying the file system and computing abstract state. To test a new file system, we need to specify only the device type (*e.g.*, RAM disk for most file systems, MTD block device for JFFS2) and the desired device size in Metis.

3.7 Limitations of Metis

False negatives. Like many other tools, Metis might experience false negatives: it could fail to detect an existing bug. First, since Metis's abstract state excludes time-related attributes, it cannot detect, *e.g.*, atime-related bugs. Though that is an unavoidable consequence of abstraction, we strive to make the abstract state as comprehensive as possible. Second, Metis identifies bugs by detecting behavioral discrepancies between the reference file system and the file system under test. Given the nature of differential testing [26, 56], Metis could fail to detect bugs shared between both file systems as no discrepancy would be found. To address this problem, one can either use a flawless reference file system or leverage N-version programming [6], comparing more than two file systems, to reduce the probability that the same bug is present across all of them. Unfortunately, a completely bug-free file system does not exist. Despite recent efforts to formally verify certain file system properties, these verified file systems may still hide bugs [14]. Furthermore, while Metis was programmed to test any number of file systems concurrently, employing a majority voting scheme on more than two adds overhead and slows exploration. (That is one reason why we support distributed verification: to increase the overall exploration rate.)

Test overhead. As Metis tracks both abstract and concrete states, it inevitably introduces extra overhead due to memory demands and the time taken for comparisons. Metis retains file system images in memory for state backtracking, although we limited memory consumption to the extent possible by choosing a minimum device size and restricting search depth. For file systems with a relatively small device-size requirement, such as Ext4 (256KiB minimum), Metis's peak memory consumption remains relatively low (2.4GiB). However, a file system with a larger minimum device size inherently consumes more memory. For example, XFS has a minimum size of 16MiB, leading to a

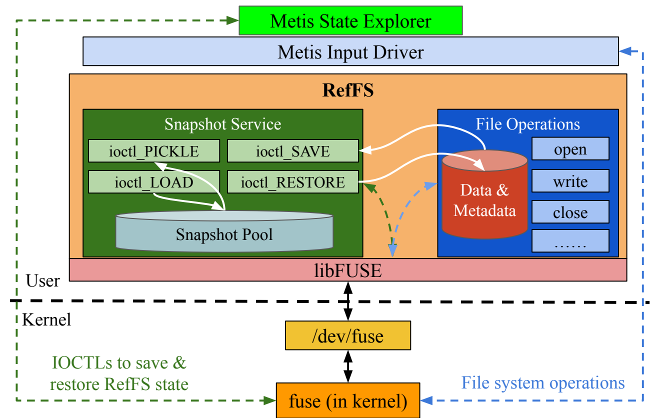


Figure 2: RefFS architecture and its interaction with Metis and kernel space. RefFS supports standard POSIX operations and provides snapshot services with a snapshot pool and four new APIs.

potential memory use of 156GiB when we use a maximum depth of 10,000. To mitigate this issue, we reduced SPIN's maximum search depth below the default 10,000, decreasing resource and memory consumption while concomitantly reducing the size of the state space. Although we experimented with memory compression (*i.e.*, zram [28]) and added swap space to increase effective memory capacity, these choices actually reduced the overall state-exploration rate. The necessity of mounting and unmounting between each operation introduces additional time overhead to Metis. Since doing so is necessary for tracking full file system states, we mitigated this cost by deploying more VTs on multiple machines and using RAM disks.

Bug detection and root-cause analysis. At present, Metis lacks the capability to identify crash-consistency and concurrency bugs in file systems. Due to the absence of crash state emulation [47, 59], Metis cannot find bugs that arise solely during system crashes. We plan to provide the option of invoking utilities such as fsck [63] between each Metis unmount/mount pair to help detect crash-consistency bugs. Given that Metis operates on file systems from a single thread, it tends to miss concurrency bugs (*e.g.*, race conditions [83]). While Metis's replayer assists in reproducing bugs, another limitation is Metis's inability to precisely identify the root cause of detected state discrepancies within the code [69].

4 RefFS: The Reference File System

In Metis, the reference file system must reliably represent correct behaviors and ensure efficiency in the file system and SS/R operations. We initially chose Ext4 as the reference file system due to its long-standing use and known robustness [55]. Still, no file system, including Ext4, is absolutely bug-free. Additionally, Ext4 lacks optimizations for model-checking state operations, limiting its suitability. We believe that a reference file system should be lightweight [14, 72], easily testable and extensible, robust, and optimized for SS/R operations in model checking. Originally, we tried to modify small in-kernel file systems (*e.g.*,

ramfs), to track their own state changes. However, capturing and restoring their entire state proved extremely challenging because the state resides across many kernel-resident data structures [5]. Consequently, we developed a new file system, called RefFS, specifically designed to function as the reference system.

RefFS architecture. RefFS is a RAM-based FUSE file system. Figure 2 shows the architecture of RefFS and its interplay with Metis and relevant kernel components. It incorporates all the standard POSIX operations supported by the Input Driver along with the essential data structures for files, directories, links, and metadata. We developed RefFS in user space to avoid complex kernel interactions and have full control over its internal states. Comprising 3,993 lines of C++ code, RefFS uses the `libFUSE` user-space library together with `/dev/fuse` to bridge user-space implementations and the lower-level `fuse` kernel module. Metis handles file system operations on RefFS in the same manner as other in-kernel file systems. Most importantly, RefFS also provides four novel snapshot APIs to manage the full RefFS file system state via `ioctl`s: `ioctl_SAVE`, `ioctl_RESTORE`, `ioctl_PICKLE`, and `ioctl_LOAD`. These are described next.

4.1 RefFS Snapshot APIs

RefFS shows how file systems *themselves* can support SS/R operations in model checking through snapshot APIs. The essence of SS/R operations lies in their ability to save, retrieve, and restore the concrete state of the file system. Although RefFS is an in-memory file system lacking persistence, it possesses a concrete state (*i.e.*, snapshot) that includes all information associated with the file system. Existing file systems like BtrFS [68] and ZFS [8], which support snapshots, can only clone (some of) the persistent state but not their in-memory states. In contrast, RefFS can capture and restore the in-memory states through its own APIs. Since RefFS stores all its data in memory, it guarantees saving and restoring the entire file system state.

Snapshot pool. The snapshot pool is a hash table that organizes all of RefFS's snapshots; the key is the current position in the search tree. The value associated with each key is a snapshot structure that saves the full file system state including all data and metadata such as the superblock, inode table, file contents, directory structures, etc. The memory overhead of the snapshot pool is low because the size of the pool is smaller than Metis's maximum search depth. Because RefFS is a simple file system, the average memory footprint for each state is just 12.5KB.

Save/Restore APIs. The `ioctl_SAVE` API causes RefFS to take a snapshot of the full RefFS state and add an entry to the snapshot pool. The `ioctl_RESTORE` does the reverse, restoring an existing snapshot from the pool. When Metis calls `ioctl_SAVE` with a 64-bit key, RefFS locks itself, copies all the data and metadata into the snapshot pool under that key, and then releases the lock. Similarly, `ioctl_RESTORE` causes RefFS to query the snapshot pool for the given key. If it is found, RefFS locks the file system, restores its full state, notifies the kernel to invalidate caches, unlocks the file system, and then discards the snapshot.

Pickle/Load APIs. Unlike other file systems, RefFS maintains concrete states by itself in the snapshot pool, so Metis does not need to keep RefFS's concrete states in its stack. To ensure good performance, RefFS's snapshot pool resides in memory. However, this means that all snapshots are lost when RefFS is unmounted, which would make it challenging to analyze and debug RefFS from a desired state. Thus, committing these snapshots to disk before Metis terminates is important to ensure they are available for post-testing analysis and debugging. Given a hash key, the `ioctl_PICKLE` API writes the corresponding RefFS state to a disk file. It can also archive the entire snapshot pool to disk. Likewise, the `ioctl_LOAD` API retrieves a snapshot from disk, loading it back into RefFS to reinstate the file system state. Using the `ioctl_PICKLE` and `ioctl_LOAD` APIs, RefFS can flexibly serialize and revert to any file system state both during and after model checking, aiding bug detection and correction. Specifically, these APIs allow RefFS to gain the same benefits as Metis's post-bug replay and processing, enabling bug reproduction from any point in a Metis run.

5 Evaluation

We evaluated the efficacy and performance of Metis and RefFS, specifically: **(1)** Does Metis have the versatility to test different input partitions compared to other testing tools? (See §5.1.) **(2)** What is Metis's performance? How does it scale with the number of VTs when using Swarm verification? (See §5.2.) **(3)** What is RefFS's performance compared to other file systems? How reliable and stable is RefFS, as Metis's reference file system? (See §5.3.) **(4)** With RefFS set as the reference file system, does Metis find bugs in existing Linux file systems? (See §5.4.)

Experimental setup. We evaluated Metis on three identical machines, trying various configurations, particularly with multiple distributed VTs. Each machine runs Ubuntu 22.04 with dual Intel Xeon X5650 CPUs and 128GB RAM. We also allocated a 128GB NVMe SSD for swap space. We evaluated Metis's performance using RAM disks, HDDs, and SSDs by comparing Ext4 with Ext2. The results showed that RAM disks were 20× faster than HDD and 18× than SSD. Also, Metis performs best when the file system device is as small as possible. Therefore, we used RAM disks as backend devices for on-disk file systems and minimum mountable device sizes for all file systems in all evaluations that follow.

5.1 Test Input Coverage

We assessed input coverage (§2) for Metis and other file system tests on two dimensions: completeness and versatility. Completeness considers whether a testing tool covers all input partitions (§2) in test cases. Versatility is the ability to tailor test cases for any desired input coverage. Metis outperforms existing checkers and a fuzzer [25] on both dimensions.

Comparison with existing testing tools. We selected three testing tools, each representing a unique technique: CrashMonkey [59] for automatic test generation, xfstests [71] for (hand-written) regression testing, and Syzkaller [25] for fuzzing.

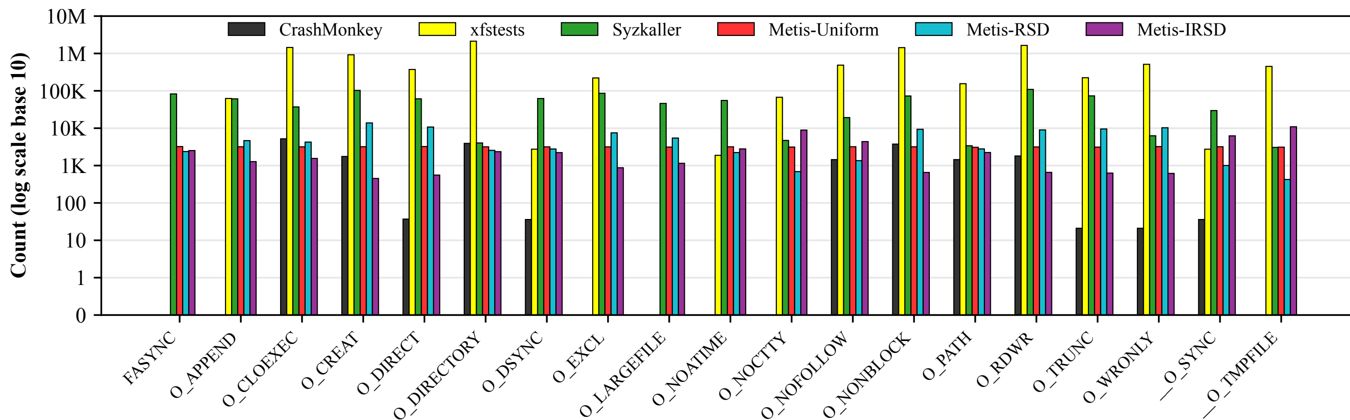


Figure 3: Input coverage counts (log_{10}, y-axis) of `open` *flags (x-axis) for CrashMonkey, xfstests, Syzkaller, and Metis with 3 different weight distributions.*

To ensure fairness, we ran all of them and Metis (with one VT) to check Ext4 for 40 minutes each, because this time length was sufficient to complete all xfstests test cases and CrashMonkey's default test cases [60].

Measuring input coverage requires tracking the file system syscalls executed by the testing tool, including their associated arguments. Traditional syscall tracers (*e.g.*, `ptrace`-based ones) cannot distinguish the syscalls used on the file systems under test, because a testing tool makes many testing-unrelated syscalls, such as opening and reading dynamically linked libraries or logging statistics. CrashMonkey and xfstests do not inherently log their test inputs. Hence, we used a tool [52] specifically designed for measuring input coverage in file system testing to assess coverage for CrashMonkey and xfstests. Syzkaller's debug option and Metis's logger record all syscalls and arguments, enabling us to compute their input coverage using their internal mechanisms.

Input coverage for open flags. Figure 3 shows the input coverage of `open`, partitioned by individual flags, for CrashMonkey, xfstests, Syzkaller, and Metis. In Metis, we set weights according to three input partition distributions: Uniform, RSD (Rank-Size Distribution [66]), and IRSD (Inverse Rank-Size Distribution [62]). Metis-Uniform denotes that Metis tests each input partition (*i.e.*, `open` flag) with a fixed weight (*i.e.*, probability). Both RSD and IRSD represent non-uniform distributions. We adopted the core principle of RSD, such that flags with higher ranks have higher test frequencies. Conversely, in IRSD, lower-ranked flags have higher frequencies. We analyzed the frequency of individual open flags' appearance in the 6.3 Linux kernel source. Metis employed those flags based on their proportional (Metis-RSD) and inverse-proportional (Metis-IRSD) frequencies. These distributions attempt to model two contrasting strategies: (1) Flags that appear more frequently in the kernel sources warrant proportionally more testing because they are used more frequently; conversely, (2) Flags with fewer occurrences in the kernel should be tested more thoroughly because they are more rarely used and hence could hide bugs for years.

In Figure 3, the x-axis labels every single-bit `open` flag and the y-axis (log_{10}) counts how often each was exercised by the testing tool. A higher y-value means more testing was conducted. We see that only Syzkaller and Metis covered all `open` flags. For instance, neither CrashMonkey nor xfstests tested the `O_LARGEFILE` flag, which could lead to missing related bugs [79]. Metis-Uniform test all flags equally; its coefficient of variation (CV) [1] (standard deviation as percentage of the mean) is only 1.2% (40-minute run). For its non-uniform test distributions, close examination of Figure 3 shows that `O_CREAT` (the most common `open` flag in the kernel source) is indeed tested most often in Metis-RSD and least in Metis-IRSD. `__O_TMPFILE`, the least-frequent flag, exhibits the opposite trend. Other tools lack the versatility to adapt their test input partitions to the desired amount of testing.

Moreover, we observed that xfstests tested certain input values (*e.g.*, `O_DIRECTORY`) millions of times while others (*e.g.*, `FASYNC`) are not tested at all. However, other tools sometimes have a higher total operation count than Metis because Metis has to unmount and remount the file system to achieve state tracking and verify state equality after each operation, slowing its syscall execution speed. Given the essential role of unmount/mount for state tracking (§3.2) and the need for state comparison (§3.3), we use Swarm verification to improve the overall operation efficiency (§3.5).

Input coverage for write size. Figure 4 shows the input coverage for the `write` size (requested byte count). The x-axis represents the log_2 of the size, corresponding to the `write` size partitions (see §3.1). For example, $x = 10$ represents all sizes from 2^{10} to $2^{11} - 1$ (or 1024–2047). The y-axis (log_{10}) shows the number of times each x bucket was tested by a given tool. Only Metis ensured complete input coverage across all `write` size partitions. All other tools primarily tested sizes under 16MiB ($x \leq 24$). Certain partitions (*e.g.*, $x = 26$) were omitted by all these tools, even though systems with many GBs of RAM are now common. As with the `open` flags above, here Metis-Uniform also assigns uniform test probabilities to each write size partition. To illustrate Metis's versatility, we chose e<u>x</u>ponentially <u>d</u>ecaying distributions for write sizes. Metis-XD prioritizes testing smaller sizes more often, because they tend

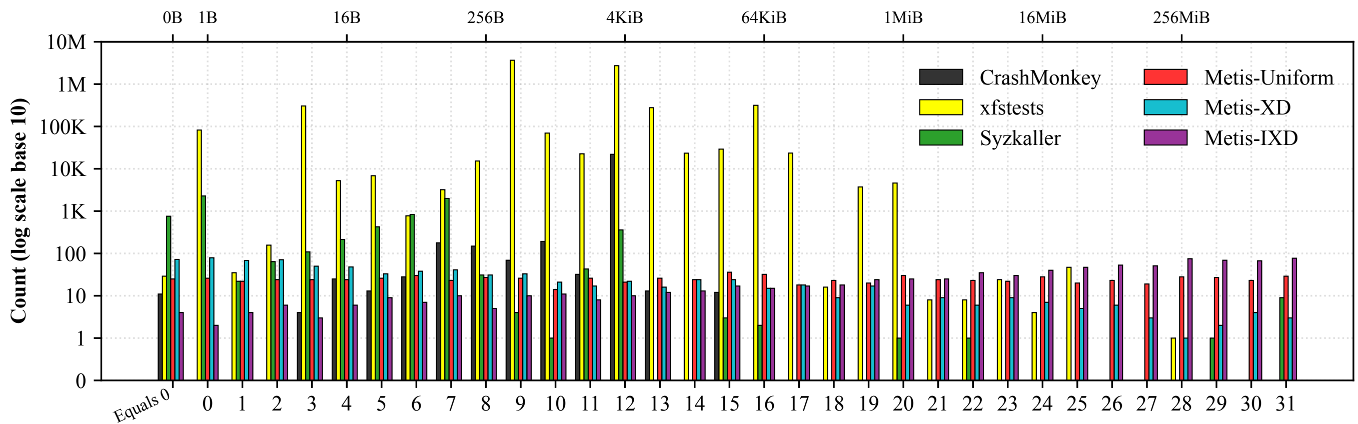


Figure 4: Input coverage (counts, $\log_{10}$, y-axis) of `write` *size (in bytes) for CrashMonkey, xfstests, Syzkaller, and Metis with three different weight distributions. The x-axis denotes the power of 2 of the write size (shown as $x2$-axis). Note a special "Equals 0" x-axis value for writes of size zero.*

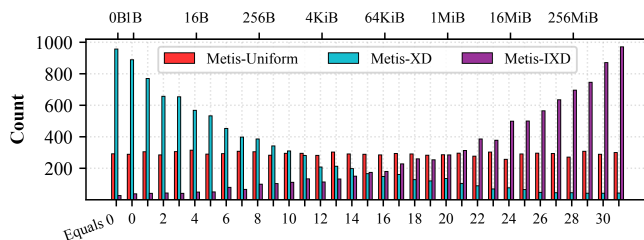


Figure 5: Input coverage of `write` *size (in bytes) for Metis-Uniform, Metis-XD, and Metis-IXD, each running for 4 hours. The x-axis and $x2$-axis here are the same as in Figure 4, but the y-axis shows counts on a linear scale. As seen, with a longer run, the expected distributions are more accurate.*

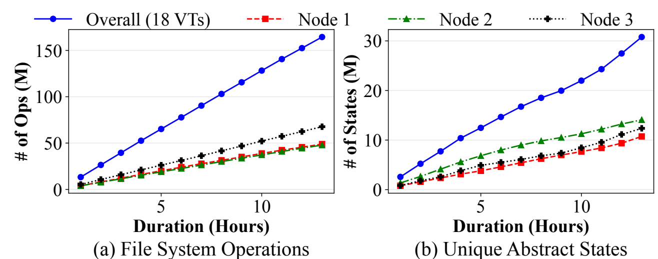


Figure 6: Metis performance with Swarm (distributed) verification, measured in terms of the number of operations and unique abstract states (in millions). Each node runs 6 VTs (one per CPU core), for a total of 18 unique VTs that collectively explored the state space. As seen, performance scales generally linearly with the number of VTs.

to be more popular in applications. The probability of each input partition is set to 0.9× smaller than the previous one (in frequency order); all probabilities are then normalized to sum to 1.0. Metis-IXD emphasizes the inverse: testing input partitions with larger write sizes, on the hypothesis that they are less used by applications and thus latent bugs may exist. Here, the probability of each test partition is 0.9× that of the next *larger* partition.

In Figure 4, the trend does not precisely align with the probabilities due to the relatively short 40-minute runtime and a correspondingly limited number of write operations, so the CV was 17.0%. When we ran Metis six times longer (4 hours), however, the CV dropped to 3.9% as seen in Figure 5; and when we ran it six times longer still (24 hours), the CV fell to a mere 2.6%. Due to space limitations, we omit showing the input coverage for other Metis-supported syscalls.

5.2 Metis Performance and Scalability

To evaluate performance with distributed Metis VTs, we deployed it on three physical nodes, comparing Ext4 (reference) to Ext2 (system under test) for 13 hours. Each node (machine) operated six individual VTs, totaling 18 VTs. Figure 6 shows the aggregate performance of the six VTs on each node, as well as the overall performance across all 18 VTs. We measured both file system operations (left) and unique abstract states (right). All VTs exhibited a linear increase in the number of operations executed over time. Over 13 hours, these 18 VTs executed more than 164 million operations, with each VT averaging 195 ops/s.

The count of explored states also increased steadily over time, although not exactly linearly. This is because executing operations does not always produce new, unseen states. For example, if a file exists, creating it again will not change the state. Thus, the number of unique states is fewer than the number of operations in a given time frame. Collectively, these VTs explored over 30 million unique states. On average, each explored 2.7 million states. Using 18 VTs resulted in exploring 11.2× more unique states than with a single VT. This experiment shows Metis's almost linear performance scalability with the number of VTs.

Different VTs might explore the same states, as each VT operates independently and without communicating with others. We evaluated the proportion of states explored by more than one VT, which represents "wasted" effort, a figure we want minimized. Our results showed that only about 1% of all states were duplicated across all VTs. Therefore, the redundancy of states explored by multiple VTs is relatively small and acceptable.

Bug#	File System	Causes & Consequences	Deterministic	Confirmed	New Bug
1	BetrFS [36]	Repeated mount and unmount caused a kernel panic	✔	✔	✔
2	BetrFS	`statfs` returned an incorrect `f_bfree`	✔	✔	✘
3	BetrFS	`truncate` failed to extend a file	✔	✔	✔
4	F2FS	A file showed the wrong size after another file was deleted	✘	✘	✔
5*	JFFS2	Data corruption occurred in a truncated file when writing a hole	✔	✔	✔
6	JFFS2	A deleted directory remained after unmounting	✘	✘	✔
7	JFFS2	GC task timeouts and deadlocks during operations	✔	✔	✘
8	JFS	NULL pointer dereference on `jfs_lazycommit`	✔	✘	✔
9	JFS	After writing to one file, another file's size changes	✘	✘	✔
10	NILFS2	NULL pointer dereference on `mdt_save_to_shadow_map`	✔	✘	✔
11	NILFS2	Failed to free space on a small device with cleaner	✔	✘	✔
12	NILFS2	Unmount operation hung after using `creat` on an existing file	✔	✘	✔

*Table 2: Kernel file system bugs discovered by Metis. This list excludes the 11 RefFS bugs that Metis detected and fixed. JFFS2 bug fix #5 (marked by *) was integrated into the Linux mainline recently.*

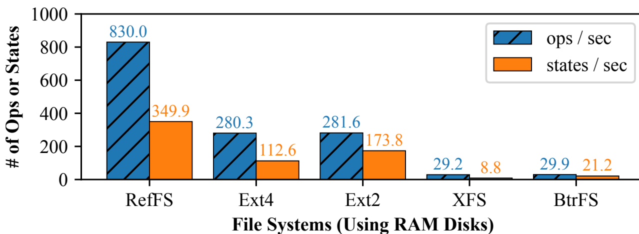


Figure 7: Performance comparison between RefFS and other mature file systems while being checked by Metis. The y-axis applies to both ops/sec and states/sec.

5.3 RefFS Performance and Reliability

To evaluate RefFS's performance, we used Metis to check it against a single file system. We also considered four other mature file systems (Ext4, Ext2, XFS, and BtrFS) as potential references. For a fair comparison, we use RAM disks as the backend devices and adopted the smallest allowed device size for each. Figure 7 shows that RefFS outperformed the others in terms of both operations and unique states per second. Even though RefFS is a FUSE file system—generally slower than in-kernel ones—it was 3.0×, 2.9×, 28.4×, and 27.7× faster than Ext4, Ext2, XFS, and BtrFS, respectively. This is primarily because Metis was able to use the save/restore APIs (§4.1) and thus did not have to unmount and remount RefFS.

Ext4 and Ext2 were faster than XFS and BtrFS due to the difference in minimum device sizes: the former require just 256KiB, whereas the latter need 16MiB. Mapping and copying larger devices in memory naturally increased time overheads.

Reliability. To serve as a reference, RefFS must be highly reliable. While developing RefFS and Metis, we made necessary changes (110 lines of code) to xfstests so that we also could use it to debug RefFS. While we used xfstests to find certain bugs in RefFS, xfstests often misreported the bug information. For example, although we implemented RefFS's `link` operation, it still did not pass generic test #2, incorrectly indicating that the operation was unsupported. For that reason, we also used Metis to check RefFS with Ext4 as the reference. We discovered and fixed 11 RefFS bugs, aided by Metis's logs and replayer. Those bugs included failure to invalidate caches, inaccurate file size updates, erroneous `ENOENT` handling, and improper updates to `nlink`, among others. After fixing them, we evaluated RefFS against Ext4 using 18 distributed Metis VTs for 30 days, executing over 3.1 billion operations and exploring 219 million unique states. No discrepancies were reported, demonstrating that RefFS's reliability and robustness are similar to Ext4's—but with better performance when used as Metis's reference file system.

5.4 Bug Finding

With RefFS as our reference file system, we applied Metis to check seven existing file systems: BetrFS [36], BtrFS [68], F2FS [49], JFFS2 [80], JFS [35], NILFS2 [18], and XFS [82], discovering potential bugs in five. Table 2 summarizes these bugs, including causes and consequences, whether they were confirmed by developers, and whether they were new or previously known. Metis found bugs using both uniform and non-uniform input distributions, but some distributions found bugs faster. Some bugs were detected within minutes, while others took up to 22 hours, which is reasonable for long-standing bugs. The bugs we identified were not detected by xfstests [71] or Syzkaller [25]. Metis identified an F2FS bug that was not detected by Hydra [43]. We also checked file systems (*e.g.*, BetrFS) that are not currently supported by Hydra [43].

We found bugs using Metis through different indicators. Discrepancies reported by the differential checker accounted for seven out of twelve detected bugs (# 2–6, 9, and 11). The remaining five caused a kernel panic (Linux "oops") or hung syscall (due to a deadlock). After analyzing each discrepancy using Metis's logger and replayer, we verified that all behavior mismatches originated from incorrect behavior in the file system under test—the reference file system, RefFS, was consistently correct.

We reported five bugs to BetrFS's and JFFS2's developers, all of which were confirmed as real bugs; however, one bug each in BetrFS and JFFS2 had already been fixed in the latest code base.

FS Testing Approach (Examples)	Input Versatility	Effort to test new FS	Effort to add new ops	State Tracking	Code Coverage Tracking	Bug Detection
Metis: this work	👍👍👍👍	👎	👎	✔	✘	Behavioral discrepancies
Traditional Model Checking: CVFS [21], CREFS [88]	👍	👎👎👎	👎👎👎	✔	✘	User-specified assertions
Implementation-level Model Checking: FiSC [87], eXplode [86]	👍	👎👎	👎👎	✔	✘	User-written checkers
Fuzzing: Syzkaller [25], Hydra [43]	👍👍	👎👎	👎	✘	✔	External checkers
Regression Testing: xfstests [71], LTP [58]	👍	👎👎	👎👎👎	✘	✘	Preset expected outcome
Automatic Test Generation: CrashMonkey [59], Dogfood [12]	👍👍	👎	👎👎	✘	✘	External checkers or an oracle

Table 3: Comparison of representative file system testing tools. In column 2, the more 👍 symbols, the more relatively versatile the system is; conversely, in columns 3–4, more 👎 symbols denote more effort.

Of the remaining unconfirmed bugs, four were deterministic and three were nondeterministic. Deterministic bugs are those easily reproducible after Metis reported a discrepancy or the kernel returned errors (*e.g.*, hang or `BUG`). We are currently pinpointing the faulty code for the deterministic bugs and preparing patches for submission to the Linux community. Metis also detected nondeterministic bugs that its replayer could not reproduce. For instance, after using `unlink` to delete file `d-00/f-01`, the size of another file `f-02` in F2FS incorrectly changed to 0 instead of the correct value. Replaying the same syscall sequence did not reproduce this bug. To trigger it, we had to rerun Metis, but the time and number of operations needed varied across experiments. Given the bug's nondeterminism, we suspect a race condition between F2FS and other kernel contexts. We verified that these unconfirmed bugs persist in the Linux kernel repository (v6.3, May 2023) without any fixes, thus classifying them as unknown bugs.

To detect them, all these potential bugs require specific operations on a particular file system state, underscoring the value of both input and state exploration. JFFS2 bug #5 is an example of the interplay between input and state. After 4.3 hours of comparing JFFS2 with RefFS, Metis reported a discrepancy due to differing file content. We observed the bug occurred when truncating a file to a smaller size, writing bytes to it at an offset larger than its size, and then unmounting the file system to clear all caches. Uncovering this multi-step, data-corruption bug required specific inputs (`truncate`, `write`) and then unmounting and remounting, because there was a cache incoherency between the JFFS2 in-memory and on-disk states. Ironically, the fact that Metis was "forced" to un/mount, is exactly why we found this bug, which was present in the 2.6.24 Linux kernel and remained hidden for 16 years. We fixed this long-standing bug, and our patch has since been integrated into the Linux mainline (all stable and development branches).

6 Related Work

File system testing and debugging. We divide existing file system testing and bug-finding approaches into five classes: Traditional Model Checking, Implementation-level Model Checking, Fuzzing, Regression Testing, and Automatic Test Generation. Table 3 summarizes these approaches across various dimensions.

Traditional model checking [21, 88] builds an abstract model based on the file system implementation and verifies it for property violations. Doing so demands significant effort to create and adapt the model for each file system, given the internal design variations among file systems [53].

Implementation-level model checking [86, 87] directly examines the file system implementation, eliminating the need for model creation. Due to file systems' complexity, however, this approach requires either intrusive changes to the OS kernel [86, 87] or manually crafting system-specific checkers [86]. Additionally, existing work [86, 87] based on this approach generally only identifies crash-consistency bugs and is incapable of detecting silent semantic bugs. Unlike these methods, Metis checks file systems for behavioral discrepancies on an unmodified kernel. Thus, there is no need to manually create checkers when testing a new file system [86]. Moreover, other model-checking approaches rely on fixed test inputs [21, 86] and lack the versatility to accommodate different input patterns. All model-checking approaches, including Metis, track file system states to guarantee thorough state exploration [15], a feature often lacking in other approaches.

Model checking and fuzzing are orthogonal approaches, each with its own advantages and disadvantages. File system fuzzing [25, 43, 83, 85] continually mutates syscall inputs from a corpus, prioritizing those that trigger new code coverage for further mutation and execution, but they cannot make state-coverage guarantees, risk repeatedly exploring the same system states, and require kernel instrumentation. Some fuzzing techniques [43, 85] also corrupt metadata to trigger crashes more easily and use library OS [65] to achieve faster and more reproducible execution than VM-based fuzzers. However, such designs have their own drawbacks: they require file-system–specific utilities to locate metadata blocks and cannot test out-of-tree file systems unsupported by library OS. Hybridra [89] enhances existing file system fuzzing with concolic execution,

but it remains fuzzing-based and has the same limitations of file system fuzzers, including the lack of state-coverage guarantees.

Fuzzing mainly supplies inputs to stress file systems and commonly finds bugs using external checkers, such as KASan [24] (memory errors) and SibylFS [67] (POSIX violations). Current fuzzers configure the tested syscalls but not their arguments [25,70], as testing is driven by code coverage. Compared to fuzzing, Metis employs a test strategy that explores both the input and state spaces, rather than solely maximizing code coverage.

Manually written regression-testing suites like xfstests [71] and LTP [58] check expected outputs and ensure that code updates do not [re]introduce bugs. Because they are hand-created, they are not easily extensible and do not attempt to automate or systematize their input or state exploration. Compared to their XFS-specific tests, xfstests' "generic" tests can be used with any file system. Nevertheless, from our past experience (including building RefFS), even when adopting the generic tests, some setup functions must be manually modified.

Automatic test generation [12, 47, 59] creates rule-based syscall workloads (*e.g.*, opening a file before writing) and employs external checkers (*e.g.*, KASan [24]) or an oracle [59] to identify file system defects. This technique is easily adapted to new file systems and extensible with new operations, owing to the universality of syscalls. Nevertheless these implementations have lacked the versatility needed to explore diverse inputs and do not explore the state space like Metis. Furthermore, these testing methods typically identify only a limited range of bugs; for instance, CrashMonkey [59] exclusively detects crash-consistency bugs. We do not include a comparative analysis of testing for other storage systems, such as NVM libraries [19] and data structures [20], given their different testing targets and goals.

Ultimately, Metis is not designed to replace any existing technique; rather, we believe that it is an additional tool that offers a complementary combination of capabilities not found elsewhere.

Verified file systems. For Metis, a reliable and ideally bug-free reference file system is critical. Verified file systems are built according to formally verified logic or specifications. For example, FSCQ [14] uses an extended Hoare logic to define a crash-safe specification and avoid crash-consistency bugs. Yggdrasil [72] constructs file systems that incorporate automated verification for crash correctness. DFSCQ [13] introduces a metadata-prefix specification to specify the properties of `fsync` and `fdatasync` for avoiding application-level bugs. SFSCQ [34] offers a machine-checked security proof for confidentiality and uses data non-interference to capture discretionary access control to preclude confidentiality bugs. However, the specifications of verified file systems have only been used to verify particular properties (*e.g.*, crash consistency [13, 14, 72] or concurrency [93]), so other unverified components can still contain bugs. Worse, even after rigorous verification, bugs can still hide due to erroneous specifications (*e.g.*, a crash-consistency bug reported on FSCQ [43]). None of these verified file systems include the extra APIs that RefFS provides, which are crucial for optimizing model-checking performance. While RefFS has not been formally verified, it relies on long-term Metis testing to attain high robustness. Thus, we chose it, rather than a verified file system, as the reference.

7 Conclusion

File system development is difficult due to code complexity, vast underlying state spaces, and slow execution times due to high I/O latencies. Many tools and techniques exist for testing file systems, but they cannot be easily updated to test specific conditions at a configurable level of thoroughness. Moreover, they tend to require code or kernel changes or cannot easily adapt to testing new file systems.

In this paper, we presented Metis, a versatile model-checking framework that can thoroughly explore file-system inputs and states. Metis abstracts file-system states into a representation that can be used to compare the file system under test against a reference one. We designed and built RefFS, a reference POSIX file system with novel features that accelerate the model-checking process. When used with Metis, RefFS is 3–28× faster than other, more established, file systems. We extensively evaluated Metis's input and state coverage, scalability, and performance. Metis, helped by RefFS, can speed file-system development: we already found a dozen bugs across several file systems. Overall, we believe that Metis, with its unique features, serves as a valuable addition to file system developers' tool suite. Finally, Metis's framework is versatile enough to be adapted to other systems (*e.g.*, databases).

Future work. Our near-term plans include expanded state exploration using Swarm verification, investigating any bugs we discover, and then fixing and reporting them. We are also beginning to test network and distributed/parallel file systems [29].

In the long run, we plan the following: (i) Metis can trigger nondeterministic bugs, such as race conditions. Therefore, we need to integrate techniques to more deterministically explore and reproduce such bugs [23]. Also, we plan to explore kernel thread interleaving states to find more concurrency bugs [83]. (ii) We intend to enhance Metis by emulating crash states to identify crash-consistency bugs in kernel file systems [47, 59]. (iii) We aim to add support for testing controlled file-system corruptions [29, 85]. For example, if both RefFS and the test file system can be corrupted in a logically identical fashion, Metis can investigate more error paths (*e.g.*, those leading to `EIO`).

Acknowledgments

We thank the anonymous FAST reviewers, our shepherd Haryadi S. Gunawi, and Dongyoon Lee for their valuable comments; and to Yizheng Jiao and Richard Weinberger for their assistance in confirming bugs in BetrFS and JFFS2. We also thank fellow students Rohan Bansal, Tejeshwar Gurram, and Shushanth Madhubalan for their contributions. This work was made possible in part thanks to Dell-EMC, NetApp, Facebook, and IBM support; a SUNY/IBM Alliance award; and NSF awards CNS-1900589, CNS-1900706, CCF-1918225, CNS-1951880, CNS-2106263, CNS-2106434, CNS-2214980, CPS-1446832, ITE-2040599, and ITE-2134840.

A Artifact Appendix

Abstract

The paper artifact contains the implementations of the *Metis* model-checking framework, the *RefFS* reference file system, and other necessary components as well as the code needed to reproduce most of the experimental results presented in this paper. Our artifact allows straightforward checking of those Linux file systems supported by Metis, and can be easily adapted to examine other file systems. We also provide documentation that explains how to set up the environment, scale up the exploration process, and detect and reproduce file system bugs based on Metis's logs and replayer.

Scope

This artifact is intended not only to validate the main claims in this paper but also to enable others to use and extend our tools, find more file-system defects, and enable future research. Specifically, we include code that automatically reproduces the results discussed in §5, including:

- Input coverage results shown in Figures 3, 4, and 5.
- Metis performance using Swarm verification in terms of operations and unique abstract states per second, as presented in Figure 6.
- RefFS performance compared to other file systems while using Metis, as shown in Figure 7.
- Detection and reproduction of file system bugs that were found by Metis.

Contents

The artifact includes two main Git repositories: the Metis file system model-checking framework and the RefFS user-space reference file system. Additionally, it contains several auxiliary Git repositories that support a basic model-checking facility and coverage analysis. Specifically, the artifact includes:

- Source to compile and execute the Metis framework for checking file systems.
- Source to build and operate the RefFS reference file system.
- Scripts to reproduce most of the experimental results appearing in this paper.
- Modified SPIN and Swarm verification scripts, optimized for seamless integration with Metis.
- The IOCov [52] tool used to compute input and output coverage for file-system testing tools.

Hosting

All the repositories are hosted on GitHub with README files for documentation; some are archived using Chameleon Cloud's Trovi service [41] and Zenodo with a permanent DOI.

Metis Repository

- Repository: *https://github.com/sbu-fsl/Metis*
- Branch: "master"
- Commit: ae08f6802be7cacb614847ebce78c18af86d553a
- Zenodo Archive [50]: *https://zenodo.org/records/10537199*
- DOI: *https://doi.org/10.5281/zenodo.10537199*

RefFS Repository

- Repository: *https://github.com/sbu-fsl/RefFS*
- Branch: "master"
- Commit: 680f5539791fc9c410d7d3cfcf2970ec4edf43a6
- Zenodo Archive [51]: *https://zenodo.org/records/10558327*
- DOI: *https://doi.org/10.5281/zenodo.10558327*

Other Repositories

- Repository of the Modified SPIN: *https://github.com/sbu-fsl/fsl-spin*
- Repository of the Modified Swarm Verification Tool: *https://github.com/sbu-fsl/swarm-mcfs*
- IOCov Repository: *https://github.com/sbu-fsl/IOCov*

Requirements

Generic Requirements

The artifact requires x86 Ubuntu 20.04 or 22.04 with one of the following Linux kernel versions: 5.4.0, 5.15.0, 5.19.7, 6.0.6, 6.2.12, 6.3.0, or 6.6.1. It may work with other Linux distributions and kernels but we did not test that.

Metis is both CPU- and memory-intensive. Running the artifact does not demand specific CPU resources, but a higher-end CPU can improve the performance of Metis's state-space exploration. Metis's memory usage depends on the type of file system being checked. Generally, the required memory size needs to be at least the sum of the minimum mountable sizes of the two file systems being compared (the file system under test and a reference file system), multiplied by Metis's maximum search width (default 10,000). Therefore, larger amounts of RAM are helpful. If sufficient RAM is not available, we recommend setting up a swap disk on a fast device such as a high-end SATA-SSD or NVMe-SSD. Metis also generates many logs during execution, so we recommend using at least a 500GB disk to avoid running out of log space.

This artifact comes with several prerequisites. We therefore provide a script `script/setup-deps.sh` in the Metis repository to automatically install all the required tools and libraries on an Ubuntu system.

Requirements for running Metis with Swarm verification

When using multiple parallel Verification Tasks (VTs) in Metis, the required computational resources amount to the demand of a single VT, multiplied by the total number of VTs. Specifically, the number of CPU cores should equal or exceed the number of VTs operating on a machine. Similarly, memory and disk resources should linearly scale with the number of VTs. The number of VTs can be configured in the `fs-state/swarm.lib` file within the Metis repository.

When VTs in Metis are distributed over multiple machines, each machine must be equipped with resources proportional to the number of VTs it runs. Moreover, in this distributed setting, one machine should be designated as the primary, with the remaining machines serving as workers. The primary machine should be set up for password-less SSH key-based access to the workers. We recommend that the hostnames of the workers are accurately entered in the `swarm.lib` configuration file on the primary machine.

References

[1] Hervé Abdi. Coefficient of variation. *Encyclopedia of Research Design*, 1(5), 2010.

[2] Alper Akcan. Fuse-ext2 GitHub repository, 2021. *https://github.com/alperakcan/fuse-ext2*.

[3] Ibrahim Umit Akgun, Geoff Kuenning, and Erez Zadok. Re-animator: Versatile high-fidelity storage-system tracing and replaying. In *Proceedings of the 13th ACM International Systems and Storage Conference (SYSTOR '20)*, pages 61–74, Haifa, Israel, June 2020. ACM.

[4] Naohiro Aota and Kenji Kono. File systems are hard to test — learning from xfstests. *IEICE Transactions on Information and Systems*, 102(2):269–279, 2019.

[5] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.10 edition, November 2023.

[6] Algirdas Avizienis. The N-Version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, 1985.

[7] Mojtaba Bagherzadeh, Nafiseh Kahani, Cor-Paul Bezemer, Ahmed E. Hassan, Juergen Dingel, and James R. Cordy. Analyzing a decade of Linux system calls. *Empirical Software Engineering*, 23:1519–1551, 2018.

[8] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The Zettabyte file system. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, March 2003. USENIX.

[9] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 83–98, Atlanta, GA, April 2016. ACM.

[10] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, July 2018. USENIX Association. Data set at *http://download.filesystems.org/auto-tune/ATC-2018-auto-tune-data.sql.gz*.

[11] Marsha Chechik, Benet Devereux, and Arie Gurfinkel. Model-checking infinite state-space systems with fine-grained abstractions using SPIN. In *International SPIN Workshop on Model Checking of Software*, pages 16–36, Toronto, ON, Canada, May 2001. Springer.

[12] Dongjie Chen, Yanyan Jiang, Chang Xu, Xiaoxing Ma, and Jian Lu. Testing file system implementations on layered models. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, pages 1483–1495, Seoul, South Korea, June 2020. ACM.

[13] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay Mert Ileri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 270–286, Shanghai, China, October 2017. ACM.

[14] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 18–37, Monterey, CA, October 2015.

[15] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. *Model Checking, 2nd Edition*. MIT Press, 2018.

[16] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pages 1–30, Elba Island, Italy, 2011. Springer.

[17] CRIU Community. Checkpoint/restore in userspace (CRIU), 2021. *https://criu.org/*.

[18] Benixon Arul Dhas, Erez Zadok, James Borden, and Jim Malina. Evaluation of Nilfs2 for shingled magnetic recording (SMR) disks. Technical Report FSL-14-03, Stony Brook University, September 2014.

[19] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 100–115, Virtual Event / Koblenz, Germany, October 2021. ACM.

[20] Xinwei Fu, Dongyoon Lee, and Changwoo Min. DURINN: adversarial memory and thread interleaving for detecting durable linearizability bugs. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 195–211, Carlsbad, CA, July 2022. USENIX Association.

[21] Andy Galloway, Gerald Lüttgen, Jan Tobias Mühlberg, and Radu I. Siminiceanu. Model-checking the Linux virtual file system. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 74–88, Savannah, GA, USA, January 2009. Springer.

[22] Bernhard Garn and Dimitris E. Simos. Eris: A tool for combinatorial testing of the Linux system call interface. In *Proceedings of the IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 58–67, Cleveland, Ohio, USA, March 2014. IEEE Computer Society Press.

[23] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 66–83, Koblenz, Germany, October 2021. ACM.

[24] Google. KASan: Linux Kernel Sanitizers, fast bug-detectors for the Linux kernel, 2023. *https://github.com/google/kernel-sanitizers*.

[25] Google. Syzkaller: Linux syscall fuzzer, 2023. *https://github.com/google/syzkaller*.

[26] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 621–631, Minneapolis, MN, USA, May 2007. IEEE Computer Society Press.

[27] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving file system reliability with I/O shepherding. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 293–306, Stevenson, WA, October 2007.

[28] Nitin Gupta. zram: Compressed RAM-based block devices, 2023. *https://www.kernel.org/doc/html/next/admin-guide/blockdev/zram.html*.

[29] Runzhou Han, Om Rameshwar Gatla, Mai Zheng, Jinrui Cao, Di Zhang, Dong Dai, Yong Chen, and Jonathan Cook. A study of failure recovery and logging of high-performance parallel file systems. *ACM Transactions on Storage (TOS)*, 18(2):1–44, 2022.

[30] Nikolas Havrikov, Alexander Kampmann, and Andreas Zeller. From input coverage to code coverage: Systematically covering input structure with k-paths. Technical report, CISPA Helmholtz Center for Information Security, 2022.

[31] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[32] Gerard J. Holzmann and Dragan Bosnacki. The design of a multicore extension of the SPIN model checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, 2007.

[33] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification techniques. *IEEE Transactions on Software Engineering*, 37(6):845–857, 2010.

[34] Atalay Mert Ileri, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Proving confidentiality in a file system using DiskSec. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 323–338, Carlsbad, CA, October 2018. USENIX Association.

[35] JFS developers. Journaled file system technology for Linux, 2011. *https://jfs.sourceforge.net/*.

[36] Yizheng Jiao, Simon Bertron, Sagar Patel, Luke Zeller, Rory Bennett, Nirjhar Mukherjee, Michael A. Bender, Michael Condict, Alex Conway, Martín Farach-Colton, et al. BetrFS: A compleat file system for commodity SSDs. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, pages 610–627, Rennes, France, April 2022. ACM.

[37] Dave Jones. Trinity: Linux system call fuzzer, 2023. *https://github.com/kernelslacker/trinity*.

[38] Nikolai Joukov, Ashivay Traeger, Rakesh Iyer, Charles P. Wright, and Erez Zadok. Operating system profiling via latency analysis. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 89–102, Seattle, WA, November 2006. ACM SIGOPS.

[39] Natalia Juristo, Sira Vegas, Martín Solari, Silvia Abrahao, and Isabel Ramos. Comparing the effectiveness of equivalence partitioning, branch testing and code reading by stepwise abstraction applied by subjects. In *Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 330–339, Montreal, QC, Canada, April 2012. IEEE Computer Society Press.

[40] Simon Kagstrom. KCOV: code coverage for fuzzing, 2023. *https://docs.kernel.org/dev-tools/kcov.html*.

[41] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the Chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, pages 219–233. USENIX Association, Virtual Event, July 2020.

[42] Kernel.org Bugzilla. Ext4 bug entries, 2023. *https://bugzilla.kernel.org/buglist.cgi?component=ext4*.

[43] 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 ACM Symposium on Operating Systems Principles (SOSP)*, pages 147–161, Huntsville, ON, Canada, October 2019. ACM.

[44] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS 2007)*, volume 1, pages 225–230, Ottawa, Canada, June 2007.

[45] Rick Kuhn, Raghu N. Kacker, Yu Lei, and Dimitris E. Simos. Input space coverage matters. *Computer*, 53(1):37–44, 2020.

[46] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*, pages 433–438, Philadelphia, PA, June 2014. USENIX Association.

[47] Hayley LeBlanc, Shankara Pailoor, Om Saran K. R. E, Isil Dillig, James Bornholt, and Vijay Chidambaram. Chipmunk: Investigating crash-consistency in persistent-memory file systems. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys)*, pages 718–733, Rome, Italy, May 2023.

[48] Doug Ledford and Eric Sandeen. Bug 513221: Ext4 filesystem corruption and data loss, 2009. *https://bugzilla.redhat.com/show_bug.cgi?id=513221*.

[49] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 273–286, Santa Clara, CA, February 2015. USENIX Association.

[50] Yifei Liu, Manish Adkar, Gerard Holzmann, Geoff Kuenning, Pei Liu, Scott Smolka, Wei Su, and Erez Zadok. Artifact package: the Metis file system model checking framework, January 2024.

[51] Yifei Liu, Manish Adkar, Gerard Holzmann, Geoff Kuenning, Pei Liu, Scott Smolka, Wei Su, and Erez Zadok. Artifact package: the RefFS reference file system for the Metis model checking framework, January 2024.

[52] Yifei Liu, Gautam Ahuja, Geoff Kuenning, Scott Smolka, and Erez Zadok. Input and output coverage needed in file system testing. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '23)*, Boston, MA, July 2023. ACM.

[53] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST '13)*, pages 31–44, San Jose, CA, February 2013. USENIX Association.

[54] 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)*, pages 218–233, Shanghai, China, October 2017. ACM.

[55] 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 Ottawa Linux Symposium (OLS)*, volume 2, pages 21–33, Ottawa, Canada, June 2007. Ottawa Linux Symposium.

[56] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[57] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 361–377, Monterey, CA, October 2015. ACM.

[58] Subrata Modak. Linux test project (LTP), 2009. http://ltp.sourceforge.net/.

[59] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 33–50, Carlsbad, CA, October 2018. USENIX Association.

[60] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Crash-Monkey: tools for testing file-system reliability, 2023. https://github.com/utsaslab/crashmonkey.

[61] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI)*, Boston, MA, December 2002. USENIX Association.

[62] Can Özbey, Talha Çolakoğlu, M Şafak Bilici, and Ekin Can Erkuş. A unified formulation for the frequency distribution of word frequencies using the inverse Zipf's law. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 1776–1780, Taipei, Taiwan, July 2023. ACM.

[63] Brandon Philips. The fsck problem. In *The 2007 Linux Storage and File Systems Workshop*, 2007. https://lwn.net/Articles/226351/.

[64] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 433–448, Broomfield, CO, October 2014. USENIX Association.

[65] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. LKL: The Linux kernel library. In *9th RoEduNet IEEE International Conference*, pages 328–333, Sibiu, Romania, 2010. IEEE.

[66] William J. Reed. On the rank-size distribution for human settlements. *Journal of Regional Science*, 42(1):1–17, 2002.

[67] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 38–53, Monterey, CA, October 2015. ACM.

[68] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[69] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–280, Dublin, Ireland, June 2009. ACM.

[70] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, pages 167–182, Vancouver, BC, Canada, August 2017. USENIX Association.

[71] SGI XFS. xfstests, 2016. http://xfs.org/index.php/Getting_the_latest_source_code.

[72] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Savannah, GA, November 2016. USENIX Association.

[73] Wei Su, Yifei Liu, Gomathi Ganesan, Gerard Holzmann, Scott Smolka, Erez Zadok, and Geoff Kuenning. Model-checking support for file system development. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '21)*, pages 103–110, Virtual, July 2021. ACM.

[74] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A study of modern Linux API usage and compatibility: What to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, pages 1–16, London, United Kingdom, April 2016. ACM.

[75] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. Semi-valid input coverage for fuzz testing. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*, pages 56–66, Lugano, Switzerland, July 2013. ACM.

[76] Theodore Ts'o. Ext4: Fix use-after-free in ext4_xattr_set_entry, 2022. https://lore.kernel.org/lkml/165849767593.303416.8631216390537886242.b4-ty@mit.edu/.

[77] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. Model checking guided testing for distributed systems. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys)*, pages 127–143, Rome, Italy, May 2023. ACM.

[78] Elaine J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703, 1991.

[79] Matthew Wilcox and Dave Chinner. XFS: Use generic_file_open(), 2022. https://github.com/torvalds/linux/commit/f3bf67c6c6fe863b7946ac0c2214a147dc50523d.

[80] David Woodhouse, Joern Engel, Jarkko Lavinen, and Artem Bityutskiy. JFFS2, 2009.

[81] Yilun Wu, Tong Zhang, Changhee Jung, and Dongyoon Lee. DEVFUZZ: automatic device model-guided device driver fuzzing. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (SP)*, pages 3246–3261, San Francisco, CA, May 2023. IEEE.

[82] XFS – high-performance 64-bit journaling file system. https://www.linuxlinks.com/xfs/. Visited February, 2021.

[83] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. KRACE: Data race fuzzing for kernel file systems. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, pages 1643–1660, Virtual Event, November 2020. IEEE.

[84] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2313–2328, Dallas, TX, October 2017. ACM.

[85] 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 (Oakland)*, pages 818–834, San Francisco, CA, May 2019. IEEE.

[86] Junfeng Yang, Can Sar, and Dawson Engler. eXplode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 131–146, Seattle, WA, November 2006. USENIX Association.

[87] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 273–288, San Francisco, CA, December 2004. ACM SIGOPS.

[88] Jingcheng Yuan, Toshiaki Aoki, and Xiaoyun Guo. Comprehensive evaluation of file systems robustness with SPIN model checking. *Software Testing, Verification and Reliability*, 32(6):e1828, 2022.

[89] Insu Yun. *Concolic Execution Tailored for Hybrid Fuzzing*. PhD thesis, Georgia Institute of Technology, December 2020.

[90] Erez Zadok, Rakesh Iyer, Nikolai Joukov, Gopalan Sivathanu, and Charles P. Wright. On incremental file system development. *ACM Transactions on Storage (TOS)*, 2(2):161–196, 2006.

[91] Andreas Zeller, Holger Cleve, and Stephan Neuhaus. Delta debugging: From automated testing to automated debugging, 2023. https://www.st.cs.uni-saarland.de/dd/.

[92] Duo Zhang, Om Rameshwar Gatla, Wei Xu, and Mai Zheng. A study of persistent memory bugs in the Linux kernel. In *Proceedings of the 14th ACM International Conference on Systems and Storage (SYSTOR)*, pages 1–6, Haifa, Israel, June 2021. ACM.

[93] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using concurrent relational logic with helpers for verifying the AtomFS file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 259–274, Huntsville, ON, Canada, October 2019. ACM.

RFUSE: Modernizing Userspace Filesystem Framework through Scalable Kernel-Userspace Communication

Kyu-Jin Cho, Jaewon Choi, Hyungjoon Kwon, and Jin-Soo Kim

Seoul National University

Abstract

With the advancement of storage devices and the increasing scale of data, filesystem design has transformed in response to this progress. However, implementing new features within an in-kernel filesystem is a challenging task due to development complexity and code security concerns. As an alternative, userspace filesystems are gaining attention, owing to their ease of development and reliability. FUSE is a renowned framework that allows users to develop custom filesystems in userspace. However, the complex internal stack of FUSE leads to notable performance overhead, which becomes even more prominent in modern hardware environments with high-performance storage devices and a large number of cores.

In this paper, we present RFUSE, a novel userspace filesystem framework that utilizes scalable message communication between the kernel and userspace. RFUSE employs a per-core ring buffer structure as a communication channel and effectively minimizes transmission overhead caused by context switches and request copying. Furthermore, RFUSE enables users to utilize existing FUSE-based filesystems without making any modifications. Our evaluation results indicate that RFUSE demonstrates comparable throughput to in-kernel filesystems on high-performance devices while exhibiting high scalability in both data and metadata operations.

1 Introduction

Traditionally, filesystems have been implemented within the OS kernel, primarily for direct-attached block devices, such as Hard Disk Drives (HDDs) or Solid State Disks (SSDs). With the advent of next-generation storage devices, there have been significant shifts in filesystem design. Since these emerging storage devices offer high performance and unique data access interfaces, there have been proposals for new filesystems specifically tailored to those innovative hardware advancements. For Non-Volatile Memory (NVM) [6], which offers low-latency performance comparable to main memory, many filesystems are designed to support Direct-Access (DAX) mode. This mode eliminates redundant memory copying and facilitates direct access to NVM [24, 26, 38, 39]. Filesystems optimized for Zoned-Namespace (ZNS) SSDs [11] actively control data placement, ensuring alignment with the device's interface that mandates sequential data writes [16, 31].

Furthermore, the explosive growth in data scale has led to the development of various distributed storage solutions. These storage platforms offer finely tuned APIs that are optimized for their internal architectures. Consequently, the customization of filesystems to enhance performance for specific workloads and platforms has become a prevalent practice [5, 8, 10, 17, 37, 41].

Yet, developing and modifying an in-kernel filesystem is challenging. Developers must possess a deep understanding of intricate kernel subsystems, including page cache, memory management, block layers, and device drivers, among others. Additionally, there is a risk of inadvertently misusing complex kernel interfaces. This inherent complexity often leads to insecure implementations of in-kernel filesystems, rendering them vulnerable to critical issues, including system crashes. In addition, efforts to integrate specialized functionalities into existing in-kernel filesystems can intensify these challenges.

Alternatively, userspace filesystems are gaining attention in both industry and academia owing to their notable advantages. They offer greater reliability and safety since programming errors won't compromise the whole system. They can also leverage mature user-level libraries and debugging tools, simplifying filesystem maintenance. Userspace filesystems are easily portable across different operating systems, in contrast to in-kernel filesystems which are intrinsically tied to a specific OS kernel interface.

FUSE [36] is a framework that allows users to develop custom filesystems without requiring kernel-level modifications. It enables filesystem operations to be implemented in userspace, making it easier to develop and maintain specialized filesystems for various purposes, including filesystems for new types of storage devices, networked or distributed filesystems, or user-specific data storage. FUSE has gained popularity for its flexibility and compatibility, making it a valuable tool for building user-level filesystem extensions.

However, FUSE is often criticized for the significant overhead it incurs due to its complex software stack. Each FUSE

request, originating from the Virtual File System (VFS) layer, must undergo multiple steps before finally reaching the userspace implementation. During this process, FUSE incurs several context switches between the kernel and userspace and memory copy overhead. Also, the single queue used by the FUSE driver to dispatch filesystem requests to the userspace FUSE daemon prevents FUSE from achieving scalable performance. These overheads become even more prominent in modern hardware environments with a large number of cores and high-performance devices.

Numerous efforts have been made to mitigate the inherent overhead in FUSE [3, 15, 23]. These approaches primarily focus on enhancing communication between the kernel and userspace, aiming for performance on par with in-kernel filesystems. However, they are only partially effective, since they share the FUSE's fundamental design that relies on a single queue. Moreover, they often require developers to either reimplement the filesystem functions or introduce new implementations, which makes them incompatible with existing FUSE-based filesystems.

In this paper, we introduce RFUSE, a novel userspace filesystem framework designed to support scalable communication between the kernel and userspace. RFUSE is specifically engineered to mitigate the overheads in FUSE's internal architecture and offers improved support for modern hardware environments. To achieve this, RFUSE leverages a ring buffer data structure, commonly used for efficient message passing, to facilitate kernel-userspace communication. RFUSE has the following three design goals:

- **Scalable kenel-userspace communication.** RFUSE employs per-core, NUMA-aware ring channels, ensuring that requests transmitted across distinct channels are delivered free from lock contention. This approach maximizes the parallelism of request processing, resulting in high scalability.
- **Efficient request transmission.** RFUSE maps the ring channels as shared memory between the kernel and userspace and uses hybrid polling to efficiently transmit requests and replies. This approach effectively reduces context switches and request copy overheads.
- **Full compatibility with existing FUSE-based filesystems.** RFUSE provides the same set of APIs as FUSE, allowing existing FUSE-based filesystems to run seamlessly on RFUSE without any modifications.

To demonstrate RFUSE's scalability in a contemporary hardware environment, we carried out a series of experiments, comparing the results with other userspace filesystem frameworks. Our evaluation shows that RFUSE effectively reduces communication latency by 53%. In addition, RFUSE exhibits significantly better performance in the majority of I/O workloads. Especially, RFUSE achieves 2.27x higher throughput than FUSE in the random read workload. Furthermore,

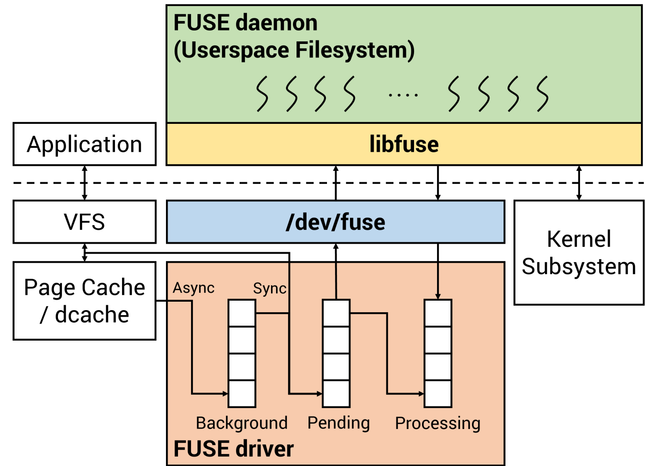


Figure 1: The internal architecture of the FUSE framework. For brevity, the *forget* queue and the *interrupt* queue are omitted in this figure.

RFUSE achieves better scalability than other frameworks in both data and metadata operations. Under the several macrobenchmarks that simulate real-world use cases, RFUSE demonstrates high performance comparable to the in-kernel filesystem. The source code of RFUSE is publicly available at `https://github.com/snu-csl/rfuse`.

The rest of the paper is organized as follows. We first present our background and motivation in Section 2. Section 3 describes the design of RFUSE and Section 4 shows the experimental results. We briefly introduce related work in Section 5 and conclude the paper in Section 6.

2 Background and Motivation

2.1 FUSE (Filesystem in Userspace)

FUSE enables unprivileged users to develop their own filesystems without modifying the kernel. Figure 1 illustrates the internal architecture of the FUSE framework. FUSE consists of two main components: the *FUSE driver* within the kernel and the userspace *FUSE daemon* created when the FUSE-based filesystem is mounted.

When the FUSE driver is loaded, it creates a particular device, */dev/fuse*, which acts as an intermediary between the Virtual File System (VFS) and the FUSE-based filesystem. Internally, the FUSE driver has five types of queues: *background*, *pending*, *processing*, *forget*, and *interrupt*. The first three queues are used to route requests for filesystem operations to the FUSE daemon. The forget queue is for interaction with the directory cache (dcache), while the interrupt queue handles interrupt requests, which are generated when the kernel needs to interrupt an ongoing filesystem operation.

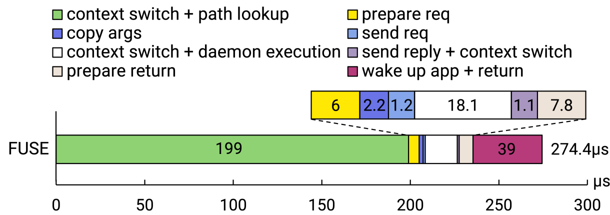


Figure 2: Latency breakdown for processing an empty filesystem operation (CREATE) in FUSE.

When applications initiate a file operation on a FUSE-based filesystem, VFS sends the request to the FUSE driver. The driver then enqueues the request in the appropriate queue depending on whether it is a synchronous or an asynchronous request. Synchronous requests are immediately added to the pending queue. In contrast, asynchronous requests, such as read-ahead or write-back requests, are initially put into the background queue before making their way to the pending queue. The FUSE driver limits the number of asynchronous requests in the pending queue to prevent interference from bulk asynchronous requests. This strategy is particularly beneficial for preserving the responsiveness of synchronous requests that are usually latency-sensitive.

When a FUSE-based filesystem is mounted, the FUSE daemon is initiated and establishes a communication channel by performing an `open()` system call on */dev/fuse*. Subsequently, the FUSE daemon creates a worker thread that performs a `read()` system call on */dev/fuse* to retrieve file operation requests. If there are no pending requests, the thread sleeps in a wait queue managed by the FUSE driver, until it receives further requests. Otherwise, the FUSE driver responds to the `read()` system call by returning the first request in the pending queue. Once the worker thread parses the request, it executes the corresponding operation according to the opcode.

A FUSE request consists of the common header, the operation-specific header, and argument(s). The common header contains the essential information required by all operations, such as the opcode and flags that denote the request's status. The operation-specific header includes the additional information specific to each operation. For metadata operations, the argument usually denotes the name of the target file(s), whereas for data operations, it indicates the required data for I/O. Both the FUSE driver and the FUSE daemon exchange these information by performing `read()` and `write()` system calls on */dev/fuse*. A FUSE reply also contains the common header and the operation-specific header. In FUSE, the headers for the request and reply are named *in_header* and *out_header*, respectively.

A FUSE daemon can have multiple worker threads. When the FUSE daemon finds no more remaining threads to receive a request from the FUSE driver, it spawns a new worker thread before handling the received request. There is no explicit limitation on the number of worker threads in FUSE, but it is implicitly controlled by the limitation imposed on the number of asynchronous requests that can reside in the pending queue.

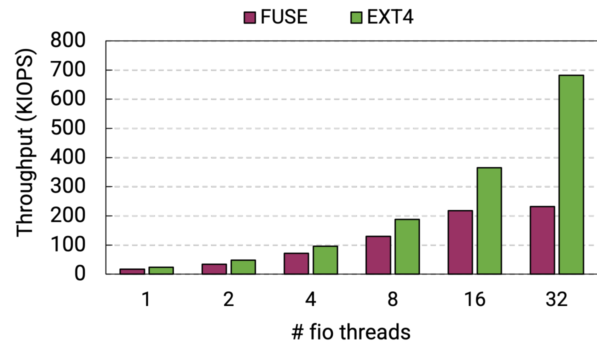


Figure 3: Scalability of random read throughput on StackFS over EXT4 (FUSE) vs. native EXT4.

2.2 Overheads in FUSE

Although FUSE provides high flexibility in developing userspace filesystems, its complex stack leads to notable performance overhead.

Latency overhead. As a first step, we conducted a latency analysis of the CREATE operation on NullFS. NullFS is a userspace filesystem we developed, which simply returns zero for any filesystem operation executed in userspace, except for the LOOKUP operation on the root directory. Figure 2 presents the latency breakdown of an empty CREATE operation, as observed in our experimental setup (see Section 4.2). The graph illustrates the various stages of the operation, highlighting the time taken at each step.

First, we can see that accessing the VFS layer and path lookup occupies 72% of the total time. Within the VFS layer, the kernel performs iterative path traversal starting from the root directory to check the existence of subdirectories and files. This path-name resolution process results in several LOOKUP operations directed towards the FUSE daemon in userspace. Hence, the latency during the initial path lookup phase (highlighted in green) encompasses the time taken for multiple rounds of context switches between the kernel and userspace. Second, the context switch and request copy overhead between the kernel driver and the FUSE daemon is not negligible. Even though NullFS does nothing but return the result, the userspace execution took as long as 18.1 μsec, due to the context switch overhead. Third, Figure 2 illustrates a significant overhead, amounting to 39 μsec, when waking up the application process that awaits a response from the FUSE daemon.

Several optimizations have been proposed to address the aforementioned latency issues in FUSE. Android 12 intro-

duced FUSE-passthrough [3] to achieve the performance of FUSE comparable to direct access to the in-kernel filesystem. With FUSE-passthrough, the FUSE driver directly forwards the READ/WRITE requests to the underlying filesystem. However, this approach bypasses the FUSE daemon, thereby sacrificing FUSE's ability to support custom userspace filesystem functions. For this reason, FUSE-passthrough is only effective for stackable filesystems that pass the unmodified requests directly to the underlying filesystem.

Another interesting approach is EXTFUSE [15]. It extends the FUSE framework, enabling the userspace filesystem to register simple eBPF [12] code snippets into the kernel. This allows various filesystem functionalities to be executed directly within a safe sandboxed environment in the kernel, avoiding costly context switches between the kernel and userspace. However, EXTFUSE requires filesystem developers to craft new functionalities within the constraints of eBPF, including limited code size, bounded loops, restricted access to kernel data, constrained pointer usage, and so on.

Bandwidth overhead and scalability issues. In FUSE, all requests from the VFS layer are placed into a shared pending queue, leading to severe lock contention, especially when multiple threads execute filesystem operations simultaneously. Not only does this design fail to harness the full bandwidth potential, but it also acts as a roadblock in the development of scalable userspace filesystems.

We ran the `FIO` benchmark to assess the scalability of random read throughput in FUSE, varying the number of `FIO` threads from 1 to 32[1]. Figure 3 contrasts the throughput of the native EXT4 filesystem with that of StackFS over EXT4. StackFS [4] is a userspace filesystem built on top of FUSE that merely passes filesystem operations to the underlying kernel filesystem (EXT4 in this experiment). Figure 3 shows that the throughput of StackFS fails to scale once the number of threads exceeds 16, while the throughput on the native EXT4 filesystem increases linearly. We note that even with a small number of threads, StackFS's bandwidth lags behind that of EXT4. We believe that the single queue-based communication in FUSE prevents StackFS from attaining scalable performance.

Recently, XFUSE [23] proposes the use of multiple communication channels to increase parallelism in FUSE. However, just adding more queues does not completely resolve the lock contention. Furthermore, the inherent context switch overhead from the original FUSE design still remains.

2.3 Motivation

Our work is inspired by io_uring [19], an efficient I/O interface introduced by the Linux kernel to address the limitations of the native asynchronous I/O interface. The io_uring interface is built around two primary elements: the Submission Queue (SQ) that holds I/O requests placed by applications, and the Completion Queue (CQ) that contains the results of those I/O requests. Typically, io_uring notifies the kernel of the submission of a new I/O request and fetches completion events from the kernel by calling the `io_uring_enter()` system call. However, io_uring offers an additional feature called *polled I/O mode* to eliminate systems calls for low latency devices. In this mode, a dedicated kernel thread monitors the submission queue while the user application polls the completion queue. The polled I/O mode enables io_uring to operate without frequently making system calls.

At its core, io_uring provides a shared memory-mapped ring buffer between the kernel and userspace to process messages to/from block devices. Using a ring buffer offers numerous advantages. First, messages (request commands or completion entries) can be enqueued into the ring buffer atomically with constant-time complexity. This capability allows the ring buffer to handle burst messages with low latency, yielding high throughput. Second, the ring buffer can be easily scaled to handle increased throughput by either enlarging its size or adding more ring buffers. Especially, a separate ring buffer can be allocated for each CPU core to minimize potential lock contention.

A comparable architecture is also employed as the communication interface between CPUs and peripheral devices. For instance, the NVMe protocol [13] utilizes a pair of ring buffers, Submission Queue (SQ) and Completion Queue (CQ), to interact with the NVMe SSDs. Similarly, Ethernet NICs (Network Interface Cards) employ Transmit (TX) and Receive (RX) ring buffers to manage outgoing and incoming network packets.

In this paper, we propose RFUSE, a novel and scalable FUSE framework that leverages a collection of ring buffers for communication between the in-kernel FUSE driver and the userspace FUSE daemon. RFUSE strives to enhance the scalability of the FUSE framework and reduce both context switch and request copy overheads by deploying ring buffer-based, per-core communication channels between the kernel and userspace. Another goal of RFUSE is to maintain the same interface as FUSE so that existing FUSE-based userspace filesystems can be executed easily over RFUSE.

3 Design

RFUSE utilizes the ring buffer structure for scalable communication between the kernel and userspace, similar to the io_uring interface. We could not directly utilize io_uring because io_uring performs request submission in the user-to-kernel direction, which does not align with the FUSE structure where kernel-to-user submission is necessary. Furthermore, as io_uring has its own kernel context, we find it challenging to facilitate flexible optimizations within the FUSE structure.

Instead, we have designed a novel *ring channel* based on a ring buffer structure, specifically to meet the needs of the FUSE framework. In this section, we delve into the mechan-

[1]The experimental setup is same as in Figure 10 (d)

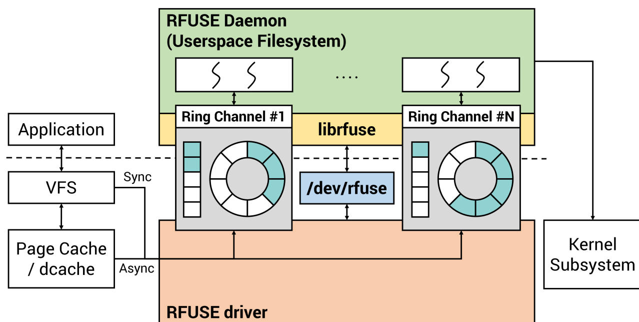


Figure 4: The overall architecture of RFUSE

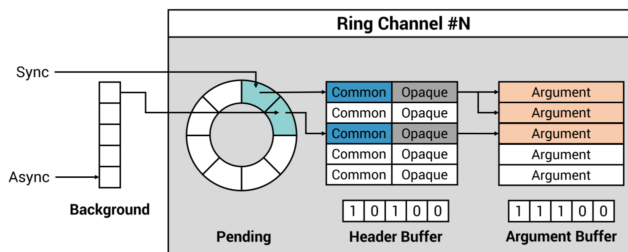


Figure 5: Components in a ring channel. For brevity, the *forget* and *interrupt* ring buffers are omitted in this figure.

ics of our ring channels and describe the design challenges associated with them.

3.1 Overall Architecture of RFUSE

RFUSE is designed to maximize performance and scalability in modern hardware environments that are equipped with many CPU cores and high-performance devices. Figure 4 depicts the overall architecture of RFUSE. Similar to FUSE, RFUSE consists of two main components: the in-kernel *RFUSE driver* and the userspace *RFUSE daemon*. However, unlike FUSE which relies on a single queue for communication between the kernel and userspace, RFUSE employs a ring channel-based message passing mechanism for each core.

When the RFUSE driver is loaded, a ring channel is created for each core in the machine along with a special device */dev/rfuse*. This architecture is intended to boost throughput by enabling parallel processing of filesystem operation requests. When a user mounts an RFUSE-based filesystem, the RFUSE daemon maps the memory region of these ring channels into the user's virtual address space using `mmap()`. This allows the userspace filesystem to exchange messages with the kernel without any context switch (see Section 3.2 for details).

When the RFUSE driver forwards a request to the RFUSE daemon, it determines the appropriate ring channel for request delivery based on the CPU core ID where the current thread is scheduled. For example, if an application thread issuing a filesystem operation runs on core 3, the RFUSE driver transmits the corresponding request to the RFUSE daemon via ring channel #3.

RFUSE allocates the memory for ring channels and their associated components in consideration of NUMA locality. When a ring channel is allocated to a different NUMA node, every access during request submission and completion incurs remote NUMA memory access penalties, resulting in substantial latency. To mitigate this, RFUSE allocates each ring channel to memory on the same NUMA node as its corresponding CPU core. This ensures that the RFUSE daemon does not access memory from a different NUMA node while processing requests.

Replacing the single queue in FUSE with per-core ring channels looks seemingly straightforward to improve performance, but it introduces several design challenges. In the following subsection, we examine the components of the ring channel and its internal operations in more detail. Section 3.3 explains how RFUSE manages worker threads on per-core ring channels. We delve into how RFUSE mitigates context switch and thread wake-up overhead through hybrid polling in Section 3.4. Section 3.5 examines RFUSE's strategies for load balancing in the face of burst asynchronous requests. Section 3.6 describes how the RFUSE daemon and the kernel driver communicate with each other using logical identifiers. The memory overhead caused by the ring channels is analyzed in Section 3.7. Lastly, Section 3.8 outlines the extensions we made in RFUSE to ensure compatibility with existing FUSE-based filesystems.

3.2 Scalable Kernel-Userspace Communication

Figure 5 illustrates the internal components of a ring channel that connects the RFUSE driver and the RFUSE daemon. Each ring channel has three ring buffers: pending, forget, and interrupt. In addition, there are two separate buffers and a background queue exists for each ring channel. Similar to FUSE, synchronous requests are enqueued directly into the pending ring buffer, while asynchronous requests are initially added to the background queue. These asynchronous requests are subsequently moved to the pending ring buffer to prevent them from exceeding the predefined maximum capacity of that buffer.

In contrast to FUSE, which sends a request in response to the system call, RFUSE utilizes a *header buffer* and an *argument buffer*. Each entry in the header buffer consists of a common header and an opaque header. The common header contains the common information for all operations such as

an opcode and a completion flag. During request submission, the opaque header holds an operation-specific header. Upon returning the result from userspace, RFUSE reuses the same header buffer entry as an out header. This approach allows RFUSE to deliver the request's outcome to the RFUSE driver efficiently.

These components of a ring channel are mapped to the virtual memory area (VMA) of the RFUSE daemon when an RFUSE-based filesystem is mounted. This establishes a shared memory space between the kernel and the RFUSE daemon. Through these shared ring buffers, the kernel can interact with the RFUSE daemon without the need to allocate and copy a request for every filesystem operation.

For example, let us consider a scenario where the VFS layer forwards a CREATE request to the RFUSE driver. The RFUSE driver first retrieves the index of an empty entry from the header buffer. Then, the driver fills the common parameter in the common header part and uses the opaque header part as *create_in_header* which is the operation-specific header of the CREATE request. Additionally, since the CREATE operation requires a filename as an argument, the driver gets a single entry from the argument buffer and records its index in the common header. After the preparation of the request, the driver enqueues the index of the header buffer entry into the pending ring buffer and increments the tail pointer. When the RFUSE daemon dequeues from the pending queue, it retrieves the index of the header buffer and parses the header to perform the appropriate userspace filesystem operation. In the case of CREATE, it returns two operation-specific out headers: *entry_out_header* containing metadata for the created file, and *open_out_header* containing file descriptor information. These are returned by reusing the opaque header and argument entry, which are used for request submission and the reply is transmitted by setting the completion flag in the common header. This approach significantly reduces the need to allocate and copy for each of requests and replies and makes efficient communication between the kernel and userspace.

RFUSE uses bitmaps for both the header buffer and argument buffer to track the allocation status of entries in these fixed-sized buffers. When all the bits in the bitmap are set, indicating that no further requests can be added to the buffer, application threads will go into a sleep state, waiting for the completion of previously submitted requests. Upon request completion, RFUSE resets the corresponding bit in the bitmap and awakens one of the threads that is in a sleep state, awaiting its turn.

3.3 Worker Thread Management

For each ring channel, the RFUSE daemon creates dedicated worker threads responsible for handling the requests received from that channel. A worker thread is bound to the corresponding CPU core by setting its CPU affinity to the same core ID as the assigned ring channel.

To completely eliminate lock contention among worker threads, it is natural to have only one worker thread per ring channel. However, this single-thread approach can negatively impact the performance. For instance, when a time-consuming operation such as FSYNC is in progress, other requests must wait until the FSYNC operation finishes. Creating as many worker threads as required, as is done in FUSE, is also not a viable option. This is because the worker threads associated with a ring channel are affinitized to the same CPU core, leading to substantial contention on that particular core.

Considering these constraints, RFUSE permits multiple workers per ring channel but caps the maximum thread count. Because the RFUSE daemon spawns only a small number of worker threads (two, by default) within a ring channel, contention on a single core remains limited. Note that there is no lock contention among worker threads operating on different ring channels.

3.4 Hybrid Polling

In FUSE, the communication between the FUSE driver and the FUSE daemon relies on `read()`/`write()` system calls on the */dev/fuse* device. When the worker threads in the FUSE daemon no longer have incoming requests to handle, or when application threads are waiting for a response from the FUSE daemon, they go into a sleep state until an event wakes them up. Using system calls leads to frequent context switches, and the event-wait mechanism between processes adds noticeable delays on the order of microseconds. This can result in significant overhead, particularly for metadata operations which typically require short latency.

Similar to the polled I/O mode in io_uring, RFUSE also supports a polling mechanism. In RFUSE, the worker threads poll the head pointer of the pending ring buffer in userspace for incoming requests, while the application threads monitor the completion flag of their submitted requests in the header buffer, waiting for a response. The use of polling eliminates not only the context switches caused by system calls, but also the delays associated with awakening threads from the sleep state. However, if polling is used in a naive manner, it can lead to the wastage of CPU resources. This inefficiency is further exacerbated in RFUSE, where both kernel and userspace threads running on the same CPU core.

As a solution, RFUSE adopts a *hybrid polling* approach. There is a user-defined period (50μsec, by default) during which a thread can perform busy-waiting idly. If the application thread in the polling state exceeds this period, it will enter the sleep state, waiting for the completion flag to be set. For requests that can be quickly handled by the userspace implementation, the application thread can receive a reply during polling and return promptly. Otherwise, for requests with longer latency, it will enter the sleep state, thus avoiding unnecessary CPU wastage. The worker threads in the RFUSE daemon also behave similarly; if there are no incoming re-

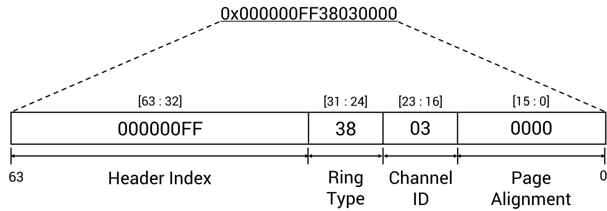


Figure 6: Encoding ring channel information in a 64-bit integer.

quests while polling the pending ring buffer, they will sleep in the wait queue.

3.5 Load Balancing of Asynchronous Requests

In the RFUSE driver, asynchronous requests are handled in a manner similar to FUSE, where they are first added to the background queue for congestion control before being transferred to the pending ring buffer. This design aims to minimize the impact of burst asynchronous requests on synchronous operations.

However, such a policy poses a problem when it is combined with RFUSE's ring selection strategy. Because RFUSE chooses the ring channel based on CPU core ID, a large number of asynchronous requests can overwhelm a single ring channel, especially for read-ahead or write-back requests that are generated in bursts by a single kernel thread. Furthermore, given the limited number of worker threads allocated to each ring, the throughput of asynchronous operations can be significantly affected. To address the skewed distribution of asynchronous requests, RFUSE implements a load-balancing policy when congestion occurs.

When enqueuing asynchronous requests into the background queue, RFUSE identifies congestion and attempts to perform load balancing based on the following two criteria: (1) when the number of requests waiting in the background queue exceeds the maximum number of asynchronous requests that can reside in the pending ring buffer, and (2) when there is a thread in the sleep state due to the prolonged execution time within the RFUSE daemon. If congestion is detected in a ring channel, RFUSE schedules the incoming asynchronous requests onto different ring channels in a round-robin fashion. This helps alleviate the load on the congested ring channel and maximize the utilization of multiple ring channels, thus increasing the overall throughput.

3.6 Transmission of Ring Channel Information

The RFUSE daemon needs to identify the locations of in-kernel data structures such as ring buffers, header buffers, and argument buffers for the following internal operations: (1) mapping the components of a ring channel in the VMA by performing `mmap()` on */dev/rfuse* during the initialization phase, (2) identifying data pages prepared for READ/WRITE requests from application threads, (3) transitioning to a sleep state on the wait queue associated with the ring buffer by `ioctl()` when the worker thread needs to stop its polling, and (4) waking up an application thread by `ioctl()` that has entered a sleeping state while waiting for completion.

However, the userspace RFUSE daemon cannot know the exact addresses of those data structures since they are allocated and managed by the kernel driver. Therefore, rather than relying on physical addresses, the RFUSE daemon utilizes logical identifiers, such as ring channel IDs, ring buffer types, and header buffer indexes, to communicate with the kernel driver. Through these logical identifiers, the userspace daemon can communicate more securely as they do not need to directly communicate via physical addresses. When the `mmap()` system call is invoked, these logical identifiers are encoded and then passed to the kernel driver using the 64-bit `offset` parameter of the `mmap()` system call.

Figure 6 depicts an example of how to encode ring channel information in a 64-bit integer. We exclude bits [15:0] due to page alignment constraints in the `offset` parameter of the `mmap()` system call. We use bits [23:16] to indicate the ID of the ring channel and bits [31:24] for ring buffer types. The remaining bits [32:63] are used to specify an entry index within the header buffer. For the `ioctl()` system call, this information is passed as the third parameter.

3.7 Memory Usage of Ring Channels

Throughout the lifespan of a userspace filesystem, ring channels remain mapped to the RFUSE daemon, retaining memory until the filesystem is unmounted. The number of ring channels matches the number of CPU cores, with both the ring buffer and the header buffer having the same number of entries. Due to some operations such as RENAME that require two arguments, the argument buffer has twice as many entries as the ring buffer. With these considerations, we can calculate the total memory usage due to ring channels as follows:

$$MemUsage = N_c \times N_r \times (S_p + S_f + S_i + S_h + 2 \times S_a) \quad (1)$$

where N_c and N_r denote the number of cores and the number of entries in the ring buffer, respectively. S_p, S_f, and S_i represent the entry size of the pending, forget, and interrupt ring buffer, respectively. Finally, S_h and S_a indicate the entry size of the header buffer and the argument buffer, respectively.

By default, RFUSE uses the following parameter values (in bytes): N_r = 4096, S_p = 4 (*integer index to the header buffer*), S_f = 32, S_i = 8, S_h = 256 (*the common and opaque header size*), and S_a = 256 (*the maximum length of the file name*). Considering an 80-core machine with 256GB of memory, the estimated memory footprint of ring channels is approximately 250MB. Given that this accounts for about 0.1% of the to-

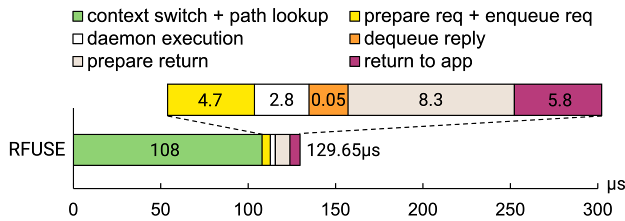


Figure 7: Latency breakdown for processing an empty filesystem operation (CREATE) in RFUSE.

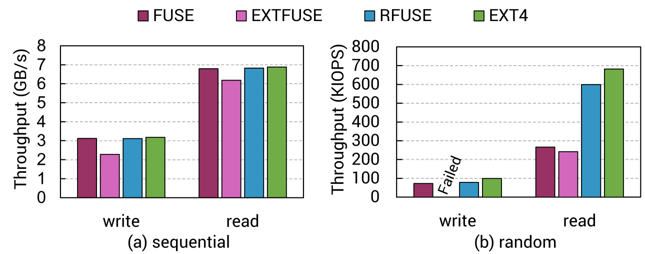


Figure 8: FIO throughput of StackFS and native EXT4.

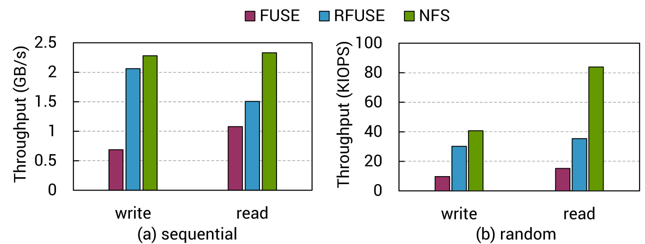


Figure 9: FIO throughput of Fuse-nfs and in-kernel NFS.

tal memory size, we believe this level of memory usage is acceptable.

3.8 Compatibility with FUSE

To make use of the ring channels, we have modified the FUSE kernel driver and the low-level layer of *libfuse* that handles message communication. In RFUSE, the READ/WRITE handlers in the kernel driver, previously used for message communication in FUSE, are now dedicated solely to data transmission for I/O requests.

Nevertheless, RFUSE retains all FUSE APIs exposed to developers of userspace filesystems. RFUSE also provides the same splicing I/O interface as FUSE, enabling data transfer between two in-kernel buffer without data copy into userspace. Thus, RFUSE ensures full compatibility with existing FUSE-based filesystems. Users do not need to rewrite their FUSE-based filesystem code when using RFUSE. The only action required is to re-link their filesystems with the *librfuse* library.

Since requests are submitted based on the CPU core ID, RFUSE requests can be executed out-of-order. Nevertheless, RFUSE ensures the same level of correctness as FUSE regarding request ordering. While FUSE utilizes a single communication queue, the userspace FUSE daemon may have multiple worker threads. This implies that simultaneous enqueuing of dependent requests may yield varying outcomes depending on the userspace filesystem implementation within FUSE. Consequently, the ordering of requests transmitted in parallel should be managed either by the VFS layer or through FSYNC-like operations initiated by applications.

4 Evaluation

4.1 Experimental Setup

Hardware setup. We used two types of testbeds to conduct our experiments. The first testbed is a Dell PowerEdge R750xs server equipped with two Intel(R) Xeon(R) Silver 4316 CPUs (80 logical cores in total) and 256GB of DDR4 memory. This testbed is also equipped with a 2TB Fadu Delta PCIe 4.0 SSD and a Mellanox ConnectX-6. Note that unless otherwise explicitly specified about the machine configuration, all experiments were carried out using this testbed. The second testbed is a Supermicro 7049GP-TRT server with two Intel(R) Xeon(R) Gold 5218R CPUs (80 logical cores in total) and 256GB of DDR4 memory. This testbed is equipped with Mellanox ConnectX-5. For the experiment on Fuse-nfs in Section 4.3.1, we used this testbed as the client and the first testbed as the server. Both testbeds run Ubuntu 20.04 LTS with the Linux kernel version 5.15.0.

FUSE frameworks tested. We conduct a comparative analysis of RFUSE against other userspace filesystem frameworks, specifically FUSE [36] v3.10.5 and the latest version of EXTFUSE [15] available on GitHub. Additionally, we have developed an emulated version of XFUSE [23], as its source code is not in the public domain. This emulation encompasses multiple FUSE communication channels corresponding to the number of CPU cores and the adaptive waiting strategy that dynamically adjusts the busy-wait period within the FUSE driver. We have excluded the RAS feature for supporting online upgrades of user-level filesystems, as it does not significantly impact filesystem performance.

User-level filesystems tested. For our experiments, we consider three userspace filesystem implementations: NullFS, StackFS [4], and Fuse-nfs [2]. To analyze and contrast the latency associated with request handling in FUSE and RFUSE, we implemented a very simple userspace filesystem called NullFS. NullFS only supports the LOOKUP operation on the root directory, and it merely returns zero for all other operations. StackFS is a stackable userspace filesystem that forwards incoming filesystem operations to an underlying

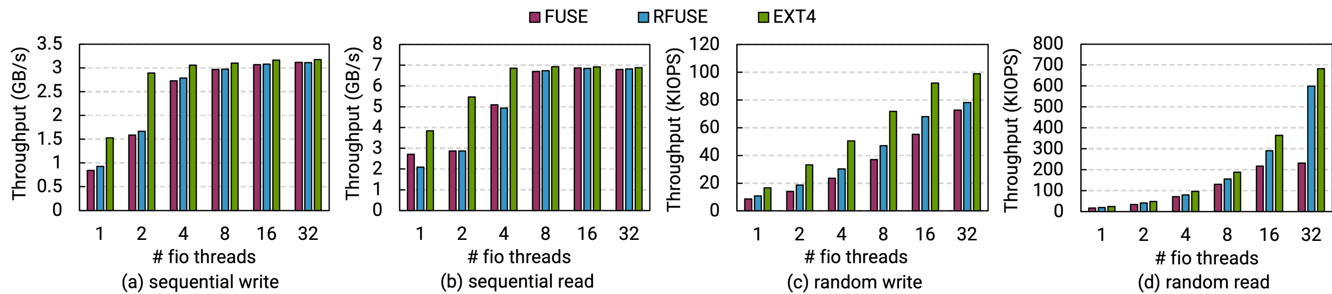


Figure 10: Scalability with FIO benchmark

in-kernel filesystem, such as EXT4. We evaluate the performance of StackFS on top of the EXT4 filesystem across three different frameworks, FUSE, EXTFUSE, and RFUSE, as well as that of the native EXT4 filesystem within the kernel. Fuse-nfs is a userspace implementation of the Network File System (NFS) client using the *libnfs* user-level library. Since EXTFUSE lacks a ported implementation of Fuse-nfs, our comparison focuses on Fuse-nfs running on FUSE and RFUSE, in addition to the in-kernel implementation of NFS.

4.2 Latency Breakdown

Figure 7 depicts the latency breakdown of a CREATE operation to NullFS on the root directory, which promptly returns without performing any action in RFUSE. In comparison to the same operation's latency in FUSE, as illustrated in Figure 2, RFUSE demonstrates a 53% lower latency. The substantial improvement in latency can be attributed to three primary factors.

First, RFUSE eliminates the need for context switches when processing requests and results. By accessing the pending ring buffer, RFUSE can retrieve the requests to be executed and quickly return the results by setting the completion flag of the corresponding entry in the header buffer. Thus, RFUSE improves the time taken in userspace by 6.46x compared to FUSE (highlighted in white in Figure 2 and Figure 7).

Second, RFUSE effectively minimizes the wake-up overhead within the kernel driver using a hybrid polling technique. After sending a request, the application thread polls the completion flag for a certain duration. Since NullFS returns the result instantly upon receiving a request, the completion is detected while the application thread is still polling, enabling immediate result retrieval.

The last factor is the improved time required for path traversal to verify the existence of subdirectories and files. As mentioned in Section 2.2, the path-name resolution initiated by the VFS layer triggers internal LOOKUP operations to the FUSE daemon along the path of the target file. Each of these LOOKUP operations results in a round trip between the kernel and userspace. Due to the reduced latency in processing a single request in RFUSE, LOOKUP operations are executed faster than in FUSE, considerably decreasing the time taken for path-name resolution.

4.3 Micro-benchmark

4.3.1 FIO Performance

To demonstrate RFUSE's ability to deliver high throughput, we perform the FIO benchmark [1] on StackFS and Fuse-nfs. The FIO benchmark is executed using 32 threads, varying both the data access pattern and the request size. For sequential I/O workloads, we use a request size of 128KB and invoke FSYNC at the end of the sequential writes. For random I/O workloads, we use a 4KB request size and trigger FDATASYNC after every write operation during random writes. Each FIO thread operates on a 4GB file with a total file size of 128GB. We also conducted the FIO benchmark using the splicing I/O interface of FUSE and RFUSE. However, we omit the results as they did not show significant differences. Note that we were unable to measure the random write throughput of EXTFUSE as it returned errors in our testing environment.

Figure 8 displays the FIO results for the native EXT4 filesystem and StackFS deployed on various frameworks. In Figure 8(a), both FUSE and RFUSE exhibit comparable throughput to EXT4 for both sequential read and write workloads. This is because, for sequential reads, the data is prefetched into the page cache through read-ahead operations, and for sequential writes, the written data is collected in the page cache before being written back in bulk. However, EXTFUSE exhibits lower throughput even for sequential workloads compared to other frameworks. EXTFUSE provides a functionality similar to fuse-passthrough, allowing I/O operations to be directly passed to the underlying filesystem via eBPF. However, this functionality was not available in the open-source version of EXTFUSE on GitHub, limiting its performance capabilities.

In Figure 8(b), RFUSE shows performance comparable to the native EXT4 filesystem for random workloads. In particular, RFUSE achieves 2.27x higher throughput than FUSE in

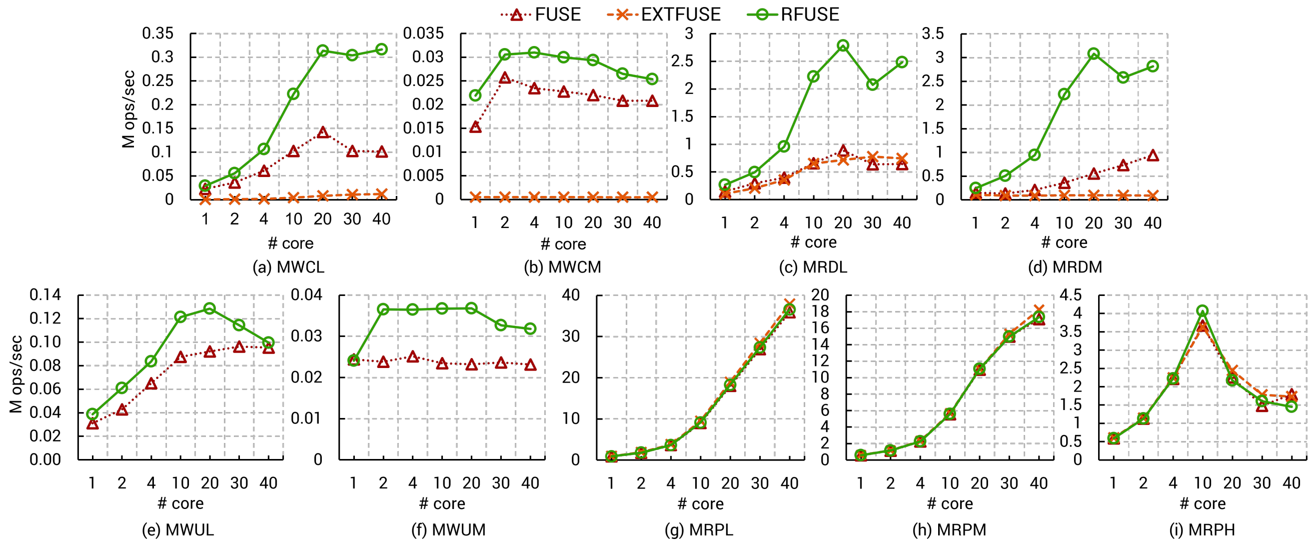


Figure 11: Scalability with FXMARK metadata operations

random reads. This is due to the effectiveness of RFUSE's hybrid polling mechanism in reducing the context switch and wake-up overhead. Considering that 4KB I/O operations typically have short execution times, FIO threads can receive the results while they are polling for them.

We conduct the same workloads on in-kernel NFS and Fuse-nfs deployed on both FUSE and RFUSE. Although it may be difficult to make a direct comparison between in-kernel NFS and Fuse-nfs due to the inherent differences in their client implementations, we consider the results from in-kernel NFS as a reference point for theoretical maximum performance. In Figure 9, RFUSE achieves higher throughput than FUSE across all workloads due to its scalable communication interface and a reduction in the average latency.

4.3.2 I/O Scalability

To investigate RFUSE's scalability compared to FUSE, we conducted experiments on StackFS with the same workloads in Section 4.3.1. We gradually increased the number of FIO threads from 1 to 32, and the results are presented in Figure 10. We have omitted the results of using splicing I/O as they followed a similar trend to those in Figure 10.

For sequential workloads, EXT4 demonstrates significantly higher throughput at lower thread counts. This can be attributed to the inherent characteristics of the FUSE and RFUSE frameworks that require communication between the kernel and userspace. Achieving sufficient throughput with fewer threads is challenging due to the communication overhead, even with the assistance of the page cache and read-ahead operations. However, when the number of threads exceeds 8, RFUSE exhibits throughput comparable to EXT4

Workload	Description
MWCL	Create empty files in a private directory
MWCM	Create empty files in a shared directory
MRDL	Enumerate a private directory
MRDM	Enumerate a shared directory
MWUL	Unlink empty files in a private directory
MWUM	Unlink empty files in a shared directory
MRPL	Open and close private files in a directory
MRPM	Open and close arbitrary files in a directory
MRPH	Open and close the same file in a directory

Table 1: Summary of metadata operation in FXMARK.

due to increased parallelism.

In Figure 10(b), we can observe that the sequential read throughput of RFUSE is lower than that of FUSE when using only one FIO thread. During the execution for sequential reads, read-ahead is performed to prefetch data. However, in RFUSE, this operation can lead to congestion on the ring channel, triggering RFUSE to initiate load balancing. Consequently, when there is only one thread, RFUSE experiences a minor performance decline due to the overhead associated with request reallocation. Nevertheless, with a higher number of threads, the increased parallelism allows RFUSE to achieve throughput comparable to EXT4.

For random workloads, RFUSE demonstrates higher throughput than FUSE while increasing the number of threads. Notably, in the random read workload, the throughput of FUSE ceased to scale beyond 16 threads, while RFUSE continues to show the scalable throughput. RFUSE exhibits better scalability due to its utilization of per-core ring channels. In addition, as mentioned in Section 4.3.1, the reduction in con-

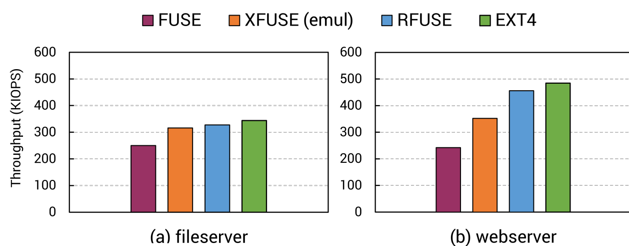


Figure 12: Thoughput of filebench workloads

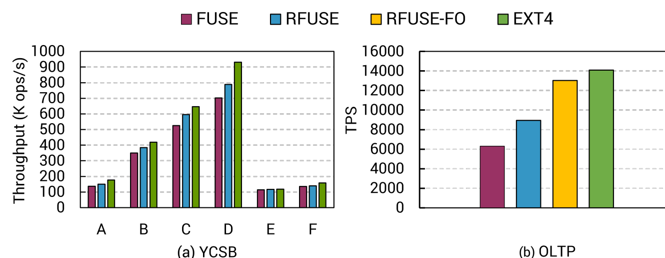


Figure 13: Thoughput of YCSB benchmark on RocksDB and sysbench OLTP benchmark on PostgreSQL

text switch and wake-up overhead enabled by hybrid polling significantly enhances RFUSE's overall performance.

4.3.3 Metadata Operation Scalability

To evaluate the scalability of RFUSE when performing metadata operations, we ran the FXMARK benchmark [29] on StackFS. Table 1 summarizes the details of the FXMARK benchmark used in the experiment. We used all the metadata workloads defined in FXMARK, with the exception of the RENAME workloads named MWRL and MWRM, as StackFS does not support RENAME operation. Note that we failed to measure the throughput of MWUL and MWUM on EXTFUSE, as it resulted in errors in our environment.

Figure 11 depicts the scalability of metadata operations for the evaluated userspace filesystem frameworks. The results show that RFUSE consistently demonstrates superior scalability compared to FUSE and EXTFUSE across the various workloads. RFUSE leverages per-core, NUMA-aware ring channels, enhancing the parallelism of metadata operations and eliminating inter-NUMA accesses, which could lead to high latency. Furthermore, RFUSE's hybrid polling proves particularly effective in metadata operations, because most of these operations can be completed quickly. This allows RFUSE to achieve both high scalability and superior throughput in workloads with contention for shared resources compared to other frameworks.

We note that EXTFUSE shows lower scalability, especially in MWCL and MWCM. It is possibly due to the key-value maps used for storing custom data structures in EXTFUSE, which may not be designed to scale effectively. For the workloads MRPL, MRPM, and MRPH, all the evaluated frameworks show similar throughput and scalability. This is because these workloads operate on a directory structure with a depth of five, where path-name resolution becomes the primary operation. As this operation heavily depends on the dcache in the VFS layer, there is little variation among the frameworks.

4.4 Macro-benchmarks

Filebench. We performed the filebench benchmark [35] on StackFS using predefined workloads, namely, fileserver and webserver, which contain a mixed set of data and metadata operations. The fileserver workload simulates the behavior of a file server that serves files to multiple clients. Files are initially created with a size of 128KB and then expanded through 16KB APPEND operations. We executed the fileserver workload using 200,000 files and 50 threads. The webserver workload mimics the behavior of a web server that serves web pages and files to clients over the Internet. Files in the webserver workload are created with a relatively small size of 16KB. We executed the webserver workload with 1.25M files using 100 threads. For both workloads, the unit size of the READ operation was set to 1MB.

We present the results of these workloads in Figure 12. In both workloads, RFUSE outperforms FUSE and XFUSE in throughput and shows performance comparable to EXT4. XFUSE exhibits superior performance compared to FUSE due to increased parallelism and its adaptive waiting strategy. However, XFUSE still suffers from context switching and request copying overhead, resulting in lower throughput compared to RFUSE. RFUSE, on the other hand, leverages communication through a ring channel, effectively eliminating these overheads and achieving higher performance than other frameworks when handling a mixed set of operations.

YCSB. To evaluate an application-level performance of each framework on a real-world workload, we deployed RocksDB [7] on StackFS and measured a throughput using the YCSB benchmark [18]. For the YCSB workloads in Figure 13(a), we initially load 50M KV pairs and run each YCSB workload with a uniform distribution. The results indicate that RFUSE can attain significant performance improvements compared to FUSE, demonstrating throughput akin to EXT4 across all YCSB workloads.

OLTP. We also deployed PostgreSQL [21] on StackFS and measured a TPS (Transactions Per Second) using the sysbench OLTP benchmark [25]. For the OLTP workload, we load 50M rows across 10 tables before running the benchmark. In Figure 13(b), RFUSE demonstrates a 42% higher TPS compared to FUSE. The results indicate that RFUSE can handle transaction processing more effectively compared to FUSE, owing to the enhanced parallelism by per-core ring

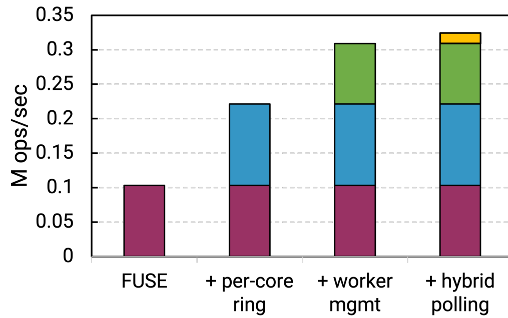


Figure 14: Impact of each technique in RFUSE

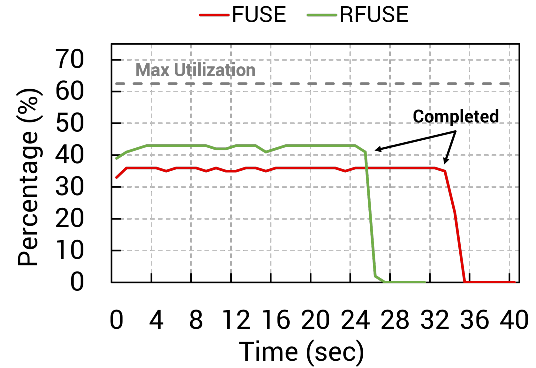


Figure 15: CPU utilization for the `fileserver` workload

channels. We can also observe that the TPS results of the frameworks are relatively lower than EXT4. This is primarily due to frequent FSYNC operations induced by logging in the `OLTP` workload. For the FSYNC operation, both FUSE and RFUSE initially writeback dirty pages of the target file to the userspace daemon and wait for the completion of all pending WRITE requests before dispatching the FSYNC request. This incurs significant processing overhead for OLTP workloads on the frameworks, leading to lower throughput compared to EXT4. To validate this, we measured the performance of RFUSE after turning off the FSYNC option in the PostgreSQL (labeled as RFUSE-FO). The result demonstrates that the TPS of RFUSE-FO is nearly on par with that of EXT4.

4.5 Factor Analysis

To assess the influence of the proposed techniques incorporated into RFUSE, we measured the throughput of the `fxmark:MWCL` workload on StackFS. Figure 14 illustrates how throughput varies as we introduce each technique one by one to FUSE. When we add per-core ring channels, we observe 2.2x higher throughput compared to the native FUSE, thanks to the enhanced parallelism. Furthermore, the management of worker threads with a CPU affinity yields a noteworthy improvement by mitigating inter-NUMA accesses. Finally, applying hybrid polling not only reduces latency but also leads to an observed improvement in throughput, while reducing contention within CPU cores.

4.6 CPU Utilization

Lastly, we measured the CPU utilization while executing the `fileserver` workload used in Section 4.4 on StackFS. Figure 15 displays the variations in CPU utilization for both FUSE and RFUSE. Considering that the `fileserver` workload operates with 50 threads on our 80-core machine, the theoretical maximum CPU utilization is 62.5%.

Owing to its hybrid polling mechanism, RFUSE exhibits roughly 7% higher CPU usage during execution compared to FUSE. However, due to RFUSE's higher throughput on the `fileserver` workload, we can see that RFUSE has a shorter execution time than FUSE. From an energy consumption perspective, despite its architecture based on hybrid polling, RFUSE is thought to consume either less or a comparable amount of energy as FUSE.

5 Related Work

Library Filesystem. A Library Filesystem (*libFS*, for short), provides a set of APIs implementing filesystem functionalities in the form of a user-level library. To access the filesystem service, applications must be directly linked to libFS during compile time. LibFS typically does not provide the standard POSIX interface. Instead, it offers filesystem APIs optimized either for specific application data access patterns or for the underlying storage platforms. Due to these benefits, many distributed filesystems are designed in the form of libFS. Examples include *libhdfs* for the Hadoop Distributed File System (HDFS) [17], *libcephfs* for CephFS [37], and many more tailored for large-scale storage systems [22, 28, 33, 41]

However, using libFS may pose some challenges. Since they do not adhere to any standardized API, applications using the POSIX API cannot directly utilize those filesystems. Also, application developers need to be familiar with the intentions and specifics of the target libFS, complicating its seamless integration with the application. Finally, a change in the filesystem API or the implementation of new funtionality require either rewriting or recompiling the application.

System Call Hooking. Several state-of-the-art NVM (Non-Volatile Memory) filesystems [14, 20, 24, 30] are implemented using a system call hooking mechanism, which allows them to directly access the NVM without going through the kernel by intercepting system calls. This is typically achieved through LD_PRELOAD [32] which is an environment variable provided by the dynamic linker,

allowing users to specify shared libraries to be loaded prior to initiating the program execution. By intercepting *libc* [9] with LD_PRELOAD, one can create a userspace filesystem using a custom library that redefines system call wrappers related to filesystem functions. However, recent studies warn about the pitfalls of implementing a userspace OS subsystem using the LD_PRELOAD hook. For example, zpoline [40] argues that LD_PRELOAD is designed to hook function calls, not system calls. System calls that are internally invoked through the `syscall` or `systenter` instruction in *libc* cannot be successfully hooked by LD_PRELOAD. This can lead to unexpected behaviors, such as FD inconsistency [27, 40], as they disrupt the synchronization between the kernel and userspace subsystems.

Restartable Userspace Filesystem. Although userspace filesystems are easy to use, a crash in a userspace filesystem remains a significant concern. Recovery from a sudden crash requires manual intervention, potentially causing disruption in services to users throughout the recovery period. Re-FUSE [34] introduces extensions into the FUSE framework for transparent and correct filesystem restart following a crash. Moreover, XFUSE [23] not only provides transparent restart capabilities but also supports online upgrades, allowing the integration of new features into the FUSE-based userspace filesystem with minimal service downtime. Such restartability feature enhances the deployment of userspace filesystems in production environments.

6 Conclusion

RFUSE is a userspace filesystem framework supporting scalable kernel-userspace communication. By harnessing per-core, NUMA-aware ring channels, RFUSE minimizes contention between worker threads and achieves high scalability. The ring channels shared between the kernel driver and the RFUSE daemon also enable RFUSE to perform efficient message transmission without the need for request copying. Moreover, a hybrid polling mechanism of RFUSE effectively reduces the costly context switches. Since RFUSE maintains the same set of APIs as FUSE, existing FUSE-based filesystems can be used without any modifications. Our evaluation results shows that RFUSE can seamlessly support modern hardware environment with its superior throughput and high scalability.

Acknowledgments

We would like to thank our shepherd, Youyou Lu, and the anonymous reviewers for their valuable feedback. This work was supported by the National Research Foundation of Korea (NRF) grant (No. 2019R1A2C2089773 and No. RS-2023-00222663), and the Institute of Information & communications Technology Planning & Evaluation (IITP) grant (No. IITP-2021-0-01363) funded by the Korea government (MSIT). This work was also supported in part by a research grant from Samsung Electronics.

References

[1] fio: Flexible I/O tester. https://fio.readthedocs.io/en/latest/fio_doc.html.

[2] fuse-nfs: A FUSE module for NFS. https://github.com/sahlberg/fuse-nfs.

[3] FUSE passthrough. https://source.android.com/docs/core/storage/fuse-passthrough.

[4] fuse-stackfs. https://github.com/sbu-fsl/fuse-stackfs.

[5] Gluster Docs. https://docs.gluster.org/en/latest/Quick-Start-Guide/Architecture/.

[6] Non-Volatime Memory (NVM). https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-memory.html.

[7] RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. https://github.com/facebook/rocksdb.

[8] S3FS: FUSE-based file system backed by Amazon S3. https://github.com/s3fs-fuse/s3fs-fuse.

[9] Standard C libraries on Linux. https://man7.org/linux/man-pages/man7/libc.7.html.

[10] SwiftFS: a userspace filesystem to mount OpenStack container in Swift. https://github.com/wizzard/SwiftFS.

[11] Zoned Namespace (ZNS) SSDs. https://nvmexpress.org/specification/nvme-zoned-namespaces-zns-command-set-specification/.

[12] eBPF: extended Berkley Packet Filter. https://www.iovisor.org/technology/ebpf, 2017.

[13] NVMe Express Base Specification. https://nvmexpress.org/specifications/, 2017.

[14] 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, 2020.

[15] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 121–134, 2019.

[16] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. ZNS: Avoiding the block interface tax for flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 689–703, 2021.

[17] Dhruba Borthakur et al. HDFS Architecture Guide. *Hadoop apache project*, 53(1-13):2, 2008.

[18] 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*, pages 143–154, 2010.

[19] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, pages 120–127, 2022.

[20] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the ZoFS user-space NVM file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 478–493, 2019.

[21] Joshua D Drake and John C Worsley. *Practical PostgreSQL*. " O'Reilly Media, Inc.", 2002.

[22] Hao Guo, Youyou Lu, Wenhao Lv, Xiaojian Liao, Shaoxun Zeng, and Jiwu Shu. SingularFS: A Billion-Scale Distributed File System Using a Single Metadata Server. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 915–928, 2023.

[23] Qianbo Huai, Windsor Hsu, Jiwei Lu, Hao Liang, Haobo Xu, and Wei Chen. XFUSE: An Infrastructure for Running Filesystem Services in User Space. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 863–875, 2021.

[24] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 494–508, 2019.

[25] Alexey Kopytov. Sysbench manual. *MySQL AB*, pages 2–3, 2012.

[26] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 460–477, 2017.

[27] James Lembke, Pierre-Louis Roman, and Patrick Eugster. DEFUSE: An Interface for Fast and Correct User Space File System Access. *ACM Transactions on Storage (TOS)*, 18(3):1–29, 2022.

[28] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 313–328, 2022.

[29] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85, 2016.

[30] Nafiseh Moti, Frederic Schimmelpfennig, Reza Salkhordeh, David Klopp, Toni Cortes, Ulrich Rückert, and André Brinkmann. Simurgh: a fully decentralized and secure NVMM user space file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[31] Myounghoon Oh, Seehwan Yoo, Jongmoo Choi, Jeongsu Park, and Chang-Eun Choi. ZenFS+: Nurturing Performance and Isolation to ZenFS. *IEEE Access*, 11:26344–26357, 2023.

[32] Kevin Pulo. Fun with ld_preload. In *linux. conf. au*, volume 153, page 103, 2009.

[33] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. 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*, pages 237–248. IEEE, 2014.

[34] 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.

[35] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[36] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of User-Space file systems. In *15th USENIX Conference on*

File and Storage Technologies (FAST 17), pages 59–72, 2017.

[37] 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.

[38] Matthew Wilcox. Add support for NV-DIMMs to ext4. https://lwn.net/Articles/613384/.

[39] 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, 2016.

[40] Kenichi Yasukata, Hajime Tazaki, Pierre-Louis Aublin, and Kenta Ishiguro. zpoline: a system call hook mechanism based on binary rewriting. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 293–300, 2023.

[41] Qing Zheng, Kai Ren, Garth Gibson, Bradley W Settlemyer, and Gary Grider. DeltaFS: Exascale file systems scale better without dedicated servers. In *Proceedings of the 10th Parallel Data Storage Workshop*, pages 1–6, 2015.

A Artifact Appendix

A.1 Abstract

RFUSE is a novel userspace filesystem framework that utilizes scalable message communication between the kernel and userspace using a per-core ring channel as a communication channel. This artifact comprises the RFUSE source code and scripts utilized in the benchmarks presented in the paper. RFUSE is implemented by modifying both the user-level library and the kernel driver of FUSE. Furthermore, this appendix provides comprehensive instructions on accessing our artifact and reproducing the results achieved in our research.

A.2 Scope

The following items represent major claims that our artifact allows to validate. For detailed descriptions and insights into the relationship between the artifact and experiments, please refer to `claims.md`

(Claim 1): *For sequential I/O operations, all frameworks and EXT4 show similar throughput due to the aid of page cache in the kernel. For random I/O operations, RFUSE demonstrates higher throughput than FUSE due to the hybrid polling mechanism in reducing context switch and wake-up overhead.*

(Claim 2): *RFUSE scales well for common data operations due to its utilization of per-core ring channels.*

(Claim 3): *RFUSE scales well for common metadata operations due to enhancing the parallelism of metadata operations and eliminating inter-NUMA accesses. For `MRPL`, `MRPM` and `MRPH` workloads, all frameworks and EXT4 show similar scalability due to the aid of dcache in the kernel.*

(Claim 4): *For `filebench` macro workloads, RFUSE outperforms FUSE and shows performance comparable to EXT4, which indicate that RFUSE is well-suited for handling a mixed set of operations.*

(Claim 5): *RFUSE demonstrates shorter latency than FUSE on NullFS due to the reduction of communication overheads.*

A.3 Contents

The submitted artifact consists of 5 components:

1. The *kernel drivers*, which contain the kernel driver codes for both FUSE and RFUSE.

2. The *user-level libraries*, which contain the user-level library for both FUSE and RFUSE.

3. The *linux kernel* source code (v5.15.0).

4. The *filesystems*, which include the source code of NullFS and StackFS.

5. The *benchmarks*, which are used in the experiments in the paper.

A.4 Hosting

The source code of RFUSE is publicly available at `https://github.com/snu-csl/rfuse` and the latest version of RFUSE is uploaded on the *master* branch.

A.5 Requirements

A.5.1 Hardware Requirements

We evaluated RFUSE on the machine equipped with Fadu Delta PCIe 4.0 SSD and 80 logical cores. For machines with older PCIe generation devices and the small number of cores, the benchmarks may not show similar results we present in the paper, but we believe the overall trends should be similar.

A.5.2 Software Requirements

We developed the RFUSE kernel driver compatible with Linux kernel version 5.15.0. To ensure the correct compilation of our artifact, please verify that your machine's kernel version matches v5.15.0.

All provided instructions are tailored for the Ubuntu OS distribution. If you intend to utilize a different Linux distribution, adjust the environment setup instructions based on the specific distribution you are using.

A.6 Set-up

This section provides concise instructions for setting up the environment and installing RFUSE from scratch. For comprehensive details including steps for mounting user-level filesystems, please refer to `README.md`.

1. Git clone our repository. The rest of the instructions assume you are in the project directory.

2. Install the Linux kernel v5.15.0 and reboot using the installed Linux kernel.

   (a) cd linux && make menuconfig

   (b) Configure `CONFIG_FUSE_FS`=m

   (c) make-kpkg –initrd –revision=1.0 kernel_image kernel_headers

   (d) cd .. && dpkg -i *.deb

   (e) Update grub to load the kernel v5.15.0 and reboot.

3. Configure the number of ring channel as the number of CPU cores in the machine.

   (a) vi lib/libfuse/include/rfuse.h driver/rfuse/rfuse.h

(b) Change the value of `RFUSE_NUM_IQUEUE` in each file to the number of cores in machine.

4. Compile and install the user library and kernel driver of RFUSE.

   (a) cd lib/librfuse && ./librfuse_install.sh

   (b) cd driver/rfuse && make && ./rfuse_insmod.sh

5. Add the location of the library to tell the dynamic link loader where to search for the library.

A.7 Experiments

For artifact evaluation, we have provided convenient scripts to execute the benchmarks used in our experiments. Please refer to `bench/README.md` for detailed instructions. Note that this guideline assumes the use of a machine with an additional storage device for conducting the experiments.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

The Design and Implementation of a Capacity-Variant Storage System

Ziyang Jiao
Syracuse University

Xiangqun Zhang
Syracuse University

Hojin Shin
Dankook University

Jongmoo Choi
Dankook University

Bryan S. Kim
Syracuse University

Abstract

We present the design and implementation of a capacity-variant storage system (CVSS) for flash-based solid-state drives (SSDs). CVSS aims to maintain high performance throughout the lifetime of an SSD by allowing storage capacity to gracefully reduce over time, thus preventing fail-slow symptoms. The CVSS comprises three key components: (1) CV-SSD, an SSD that minimizes write amplification and gracefully reduces its exported capacity with age; (2) CV-FS, a log-structured file system for elastic logical partition; and (3) CV-manager, a user-level program that orchestrates system components based on the state of the storage system. We demonstrate the effectiveness of CVSS with synthetic and real workloads, and show its significant improvements in latency, throughput, and lifetime compared to a fixed-capacity storage system. Specifically, under real workloads, CVSS reduces the latency, improves the throughput, and extends the lifetime by 8–53%, 49–316%, and 268–327%, respectively.

1 Introduction

Fail-slow symptoms where components continue to function but experience degraded performance [16, 52] have recently gained significant attention for flash memory-based solid-state drives (SSDs) [40, 41, 66]. In SSDs, such degradation is often caused by the SSD-internal logic's attempts to correct errors [3, 16, 44, 50]. Recent studies have demonstrated that fail-slow drives can cause latency spikes of up to 3.65× [40], and since flash memory's reliability continues to deteriorate over time [25, 40, 66], we expect the impact of fail-slow symptoms on overall system performance to increase.

Figure 1 demonstrates a steady performance degradation for a real enterprise-grade SSD. We age the SSD through random writes by writing about 100 terabytes of data each day, and during morning hours when no other jobs are running, we measure the throughput of the read-only I/O, both sequential and random reads. As shown in Figure 1, the performance of the SSD degrades as the SSD wears out, at a rate of 4.2% and

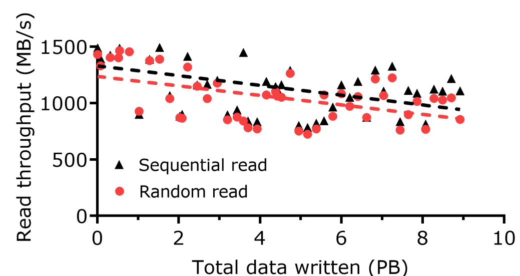


Figure 1: SSD performance degradation due to wear-out. The dashed line represents the linear regression of the daily data points. The throughput decreases by 37% for random reads and 38% for sequential reads after 9 petabytes of data writes.

4.3% of the initial performance for each petabyte written, for random reads and sequential reads, respectively. It is unlikely that the throughput drop is due to garbage collection as (1) this was measured daily over months, and (2) only reads are issued during measurement. By the end, writing a total of 9 petabytes of data to the SSD decreased the throughput by 37% for random reads and 38% for sequential reads.

To address this problem, we start with two key observations. First, flash memory, when it eventually fails, does so in a fail-partial manner. More specifically, an SSD's failure unit is an individual flash memory block [3, 44, 50], and the SSD-internal wear leveling algorithms are artifacts to emulate a hard disk drive-like fail-stop behavior [25, 31]. Second, an SSD has no other choice but to trade performance as flash memory's reliability deteriorates, because a storage device's capacity remains fixed and unchanged from its newly installed state until its retirement. SSD's internal data re-reads [4, 5, 42, 53] or preventive re-writes [6, 18] are such choices that lead to fail-slow symptoms [30, 31].

Based on the two key observations, we propose a capacity-variant storage system (CVSS) that maintains high performance even as SSD reliability deteriorates. In CVSS, the logical capacity of an SSD is not fixed; instead, it gracefully reduces the number of exported blocks below the original

capacity by mapping out error-prone blocks that would exhibit fail-slow behavior and hiding them from the host. This approach is enabled by the SSD's ability to update data out-of-place. Surprisingly, we find that maintaining a fixed-capacity interface comes at a heavy cost, and reducing capacity counterintuitively extends the lifetime of the device. Our experiments show that, compared to traditional storage systems, the capacity-variant approach of CVSS outperforms by 49–316% and outlasts by 268–327% under real-world workloads.

We enable capacity variance by designing kernel-level, device-level, and user-level components. The first component is a file system (CV-FS) that dynamically tunes the logical partition size based on the aged state of the storage device. CV-FS is designed to reduce capacity in an online, fine-grained manner and carefully manage user data to avoid data loss. The device-level component, CV-SSD, maintains its performance and reliability by mapping out aged and poor-performing blocks. Without needing to maintain fixed capacity, CV-SSD simplifies flash management firmware, avoids fail-slow symptoms, and extends its lifetime. Lastly, the user-level component, CV-manager, provides necessary interfaces to the host for capacity variance. Users can set performance and reliability requirements for the device through commands, and the CV-manager then adaptively orchestrates CV-FS and the underlying CV-SSD.

The contributions of this paper are as follows.

- We present the design of a capacity-variant storage system that relaxes the fixed-capacity abstraction of the storage device. Our design consists of user-level, kernel-level, and device-level components that collectively allow the system to maintain performance and extend its lifetime. (§ 3)
- We develop a framework that allows for a full-stack study on fail-slow symptoms in SSDs over a long time, from start to failure. This framework provides a comprehensive model of SSD internals and aging behavior over the entire lifetime of SSDs[1]. (§ 4)
- We evaluate and quantitatively demonstrate the benefits of capacity variance using a set of synthetic and real-world I/O workloads throughout the SSD's *entire* lifetime. Capacity variance avoids the fail-slow symptoms and can significantly extend the SSD's lifetime. (§ 5)

2 Background and Motivation

We first show the increasing trend of flash memory errors in SSDs and describe how flash cells wear out. We then explain how the current storage system abstraction exacerbates reliability-related performance degradation, and summarize prior work for addressing these fail-slow symptoms.

Flash memory errors and wear-out. The rapid increase

[1]Our framework and extensions are available at https://github.com/ZiyangJiao/FAST24_CVSS_FEMU

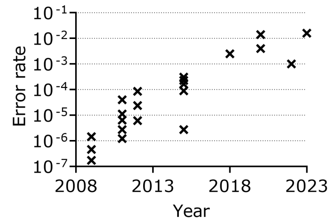


Figure 2: Flash memory error rates have increased significantly over the past years.

in NAND flash memory density has come at the cost of reduced reliability and exacerbated fail-slow symptoms. Figure 2 shows the reported flash raw bit error rates (RBERs) in recent publications [3, 4, 14, 30, 42, 55, 57, 65], and this trend indicates that flash memory errors are already a common case.

One of the significant flash memory error mechanisms is wear-out, where flash cells are gradually damaged with repeated programs and erases [44, 50]. Because wear-outs are irreversible, once a flash block reaches its endurance limit or returns an operation failure, it is marked as bad by the SSD-internal flash translation layer (FTL) and taken out of circulation. To replace these unusable blocks, SSDs are often over-provisioned with more physical capacity than the logically exported capacity.

SSD's wear-outs are caused not only by write I/Os, but also by SSD-internal management such as garbage collection, reliability management, and wear leveling (WL). Although much of the literature has emphasized the role of garbage collection in the SSD's internal writes, studies have revealed that SSD's reliability management and WL also significantly impact the lifetime [25, 27, 30, 43]. WL, in particular, is revealed to be far from perfect, wearing out some of the blocks 6× faster [43] and often leads to counterintuitive acceleration of wear-outs, increasing the write amplification factor as high as 11.49 [25].

Fixed capacity abstraction. Unfortunately, the current storage system abstraction of fixed capacity requires SSDs to implement wear leveling (WL), even if it is imperfect and harmful [25, 43]. Specifically, with the fixed capacity abstraction, the device is not allowed to have part of its capacity fail (i.e., wear out) prematurely, and therefore the SSD has to perform wear leveling to ensure that most, if not all, of its capacity is wearing out at roughly the same rate. If the SSD cannot maintain its original exported capacity when too many blocks become bad, then the entire storage device becomes unusable [50]. This is despite the fact that the SSD internally has a level of indirection and abstracts the physical capacity.

However, the file system provides a file abstraction to the user-level applications, and this abstraction hides the notion of capacity. While utility programs such as `df` and `du` report the storage capacity utilization, file operations such as `open()`, `close()`, `read()`, and `write()` do not expose capacity directly. Instead, the file system manages the storage capacity using persistent data structures such as superblock and allocation maps to track the utilization of the SSD.

The fixed capacity abstraction used between the file system and storage devices necessitates the implementation of WL

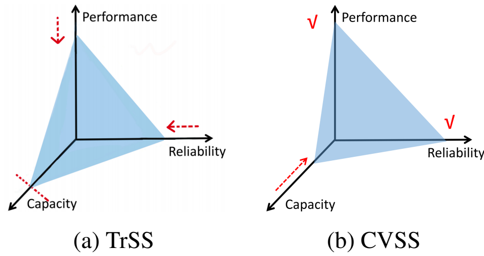


Figure 3: Comparison between the traditional fixed-capacity storage system (TrSS) and capacity-variant storage system (CVSS). For TrSS (Figure 3a), the performance and reliability degrade as the device ages to maintain a fixed capacity; for CVSS (Figure 3b), the performance and reliability are maintained by trading capacity.

on physical flash memory blocks. However, WL leads to an overall increase in wear on the SSD, resulting in a significantly higher error rate as all the blocks age. This, in turn, manifests into fail-slow symptoms in SSDs.

Fail-slow symptoms. Fail-slow symptoms are caused by the SSD's effort to correct errors [4, 5, 42, 53] and prevent the accumulation of errors [6, 18]. Because SSDs are commonly used as the performance tier in storage systems where the identification and removal of ill-performing drives are critical, fail-slow symptoms in SSDs have gained significant attention recently. Prior research in this area can be categorized into three types. The first group focuses on developing machine learning (ML) models to quickly identify SSDs experiencing fail-slow symptoms [7, 22, 61, 66, 67]. Various models, including neural networks [22], autoencoders [7], LSTM [67], feature ranking [61], and random forest [66], have been explored with varying accuracy and efficacy. The second group aims to isolate and remove ailing drives using mechanisms deployed in large-scale systems, identified through ML [40] or system monitoring [21, 52]. The third group proposes modifications to the interface to reject slow I/O and send hedging requests to a different node [20] or drive [38].

Unfortunately, ML-based learning of SSD failures requires an immense number of data points, is often expensive to train, and is only available in large-scale systems [40, 66]. Furthermore, as SSDs evolve and new error mechanisms emerge (e.g., lateral charge spreading [36] and vertical and horizontal variability [56]), older ML models become obsolete, making it difficult to reap the benefits of fail-slow prediction. Most critically, these prior approaches only treat the symptoms and fail to consider the underlying cause: the flash error mechanism.

3 Design for Capacity Variance

The high-level design principle behind the capacity-variant system is illustrated in Figure 3. This system relaxes the fixed-capacity abstraction of the storage device and enables a better tradeoff between capacity, performance, and reliability. The traditional fixed-capacity interface, which was designed

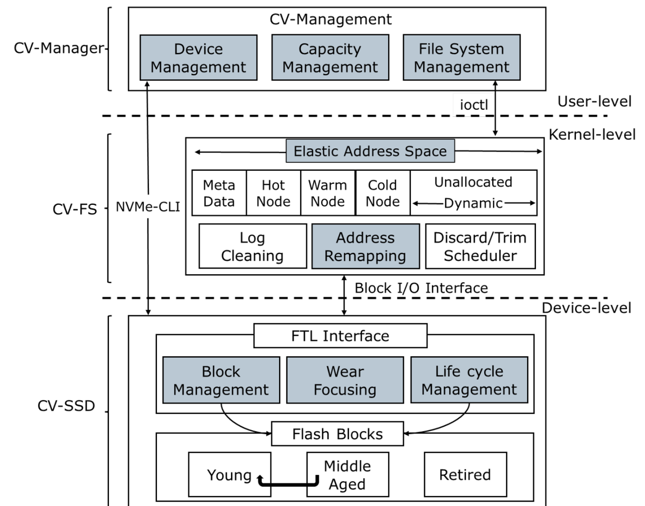
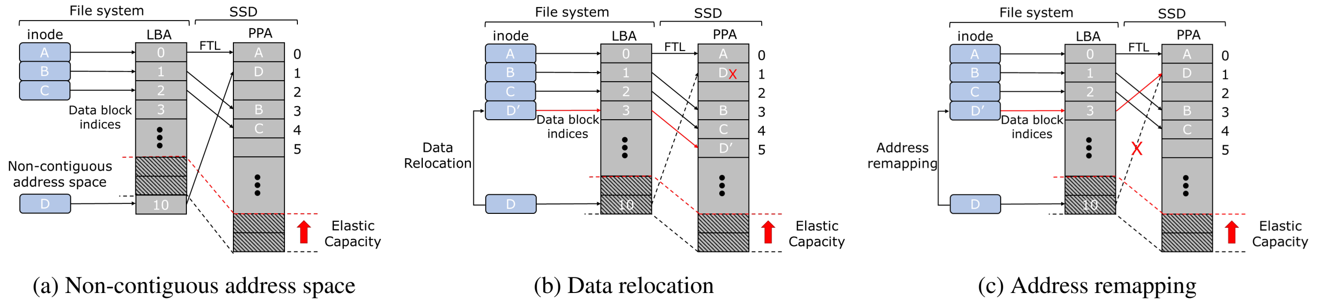


Figure 4: An overview of the capacity-variant system: (1) CV-FS exports an elastic logical space based on CV-SSD's aged state; (2) CV-SSD retires error-prone blocks to maintain device performance and reliability; and (3) CV-manager provides user-level interfaces and orchestrates CV-SSD and CV-FS. The highlighted components are discussed in detail.

for HDDs, assumes a fail-stop behavior where all storage components either work or fail at the same time. However, this assumption is not accurate for SSDs since flash memory blocks are the basic unit of failure, and it is the responsibility of the FTL to map out failed, bad, and aged blocks [31, 50].

By allowing a flexible capacity-variant interface, an SSD can gracefully reduce its exported capacity, and the storage system as a whole would reap the following three benefits.

- **Performant SSD even when aged.** An SSD can avoid fail-slow symptoms by gracefully reducing its number of exported blocks. Error management techniques such as data re-reads [4, 5, 42, 53] and data re-writes [6, 18] would be performed less frequently as blocks with high error rates can be mapped out earlier. This, in turn, reduces the tail latency and lowers the write amplification, making it easier to achieve consistent storage performance.
- **Extended lifetime for SSD-based storage.** An SSD's lifetime is typically defined with a conditional warranty restriction under *DWPD (drive writes per day)*, *TBW (terabytes written)*, or *GB/day (gigabytes written per day)* [58]. With the fixed capacity abstraction, the SSD reaches the end of its lifetime when the physical capacity becomes less than the original logical capacity. Instead, with capacity variance, the lifetime of an SSD would be extended significantly, as it would be defined by the amount of data stored in the SSD, not by the initial logical capacity.
- **Streamlined SSD design.** By adopting the approach of allowing the logical capacity to drop below the initial value, SSD vendors can design smaller and more efficient error correction hardware and their SSD-internal firmware: There



(a) Non-contiguous address space (b) Data relocation (c) Address remapping

Figure 5: Design options for capacity variance. In Figure 5a, the FS internally maps out a range of free LBA from the user, causing address space fragmentation. In Figure 5b, the data block is physically relocated to lower LBA. This approach maintains the contiguity of the entire address space but exerts additional write pressure on the SSD. Lastly, in Figure 5c, the data block can be logically remapped to lower LBA. This approach incurs negligible system overhead by introducing a special SSD command to associate data with a new LBA.

is no need to overprovision the SSD's error handling logic or to ensure that all blocks wear out evenly.

Figure 4 illustrates the main components of a capacity-variant system. Enabling the capacity variance feature is achieved by designing the following three components: (1) CV-FS, a log-structured file system for supporting elastic logical partition; (2) CV-SSD, a capacity-variant SSD that maintains device performance and reliability by effectively mapping out aged blocks; and (3) CV-manager, a capacity management scheme that provides the interface for adaptively managing the capacity-variant system.

3.1 Capacity-Variant FS

The higher-level storage interfaces, such as the POSIX file system interface, allow multiple applications to access storage using common file semantics. However, to support capacity variance, the file system needs to be modified. In this section, we discuss the feasibility of an elastic logical capacity based on existing storage abstractions and then investigate different approaches for supporting capacity variance, Lastly, we describe our new interface for capacity-variant storage systems based on the selected approach.

3.1.1 Feasibility of Elastic Capacity

Current file systems assume that the capacity of the storage device does not change and tightly couple the size of the logical partition to the size of the associated storage device. To overcome this limitation, the CV-FS file system declares the entire address space for use at first and then dynamically adjusts the declared space as the storage device ages in an online manner. This is achieved by defining a variable logical partition that is independent of the physical storage capacity.

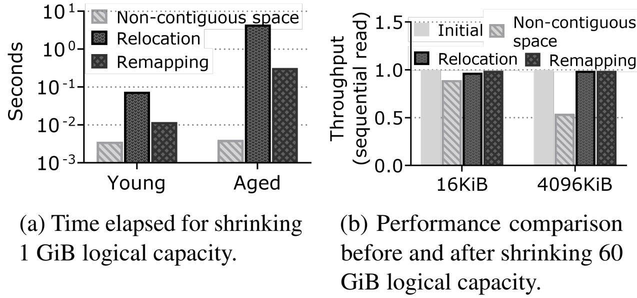
Thankfully, this transition is feasible for three reasons. First, the TRIM [47] command, which is widely supported by interface standards such as NVMe, enables the file system to explicitly declare the data that is no longer in use. This allows the SSD to discard the data safely, making it possible to reduce the exported capacity gracefully. Second, modern file systems can safely compact their content so that the data in use are contiguous in the logical address space. Log-structured file systems [54] support this more readily, but file system defragmentation [59] techniques can be used to achieve the same effect in in-place update file systems. Lastly, the file abstraction to the applications hides the remaining space left on storage. A file is simply a sequence of bytes, and file system metadata such as utilization and remaining space is readily available to the system administrator.

3.1.2 File System Designs for Capacity Variance

Shrinking the logical capacity of a file system can be a complex procedure that may result in data loss if not done carefully [31]. Most importantly, any valid data within the to-be-shrunk space must be relocated and the process must be coordinated with underlying storage accordingly. Moreover, to ensure users do not need to unmount and remount the device, the logical capacity should be reduced in an online manner, and the time it takes to reduce capacity should be minimal with low overhead.

Figure 5 depicts three approaches to performing online address space reduction: (1) through a non-contiguous address space; (2) through data relocation; and (3) through address remapping. We describe each approach and our rationale for choosing the address remapping (Figure 5c).

- **Non-contiguous address space** (Figure 5a). The file system internally decouples the space exported to users from the LBA. When logical capacity should be reduced, the file system identifies an available range of free space from the end of the logical partition and then restricts the user from using it, for example, by marking that as allocated. With this approach, the adjustment of logical capacity can be efficiently achieved with minimal upfront costs, as the primary task involved is allocating the readily available free space. However, this approach increases the file system cleaning overhead and fragments the file system address



(a) Time elapsed for shrinking 1 GiB logical capacity.

(b) Performance comparison before and after shrinking 60 GiB logical capacity.

Figure 6: Performance results for three capacity variance approaches. The address remapping approach introduces lower overhead (Figure 6a) and does not incur fragmentation after shrinking the address space (Figure 6b).

space. Due to the negative effect of address fragmentation [12, 13, 19, 24], we avoid this approach despite the lowest upfront cost.

- **Data relocation** (Figure 5b). Similar to segment cleaning or defragmentation, the file system relocates valid data within the to-be-shrunk space to a lower LBA region before reducing the capacity from the higher end of the logical partition. This approach maintains the contiguity of the entire address space. Nevertheless, it is essential to note that data relocation exerts additional write pressure on the SSD and the overhead is proportional to the amount of valid data copied. Moreover, user requests are potentially stalled during the relocation process.
- **Address remapping** (Figure 5c). Data is relocated logically at the file system level without data relocation at the SSD level by taking advantage of the already existing SSD-internal mapping table [49, 68]. While this approach necessitates the introduction of a new SSD command to associate data with a new LBA, it effectively mitigates address space fragmentation and incurs negligible system overhead, as no physical data is actually written.

We implement the three approaches above on F2FS and measure the elapsed time for reducing capacity by 1 GiB. The reported results represent an average of 60 measurements. On average, each measurement resulted in the relocation or remapping of 0.5 GiB of data for the aged file system case and 0.05 GiB of data for the young case. We further compare the performance under the sequential read workload with two I/O sizes (i.e., 16 KiB and 4096 KiB) before and after capacity is reduced. As depicted in Figure 6, the elapsed time required to shrink 1 GiB of logical space on an aged file system is 0.317 seconds when employing the address remapping approach. In contrast, the data relocation approach takes approximately 4.5 seconds. Notably, while the non-contiguous address space approach only takes 0.004 seconds, it exhibits significant performance degradation after the capacity reduction, for example, 13% for 16 KiB read and 50% for 4096 KiB read, due to increased fragmentation. We next present the design details of the proposed remapping interface and the capacity reduction process with that.

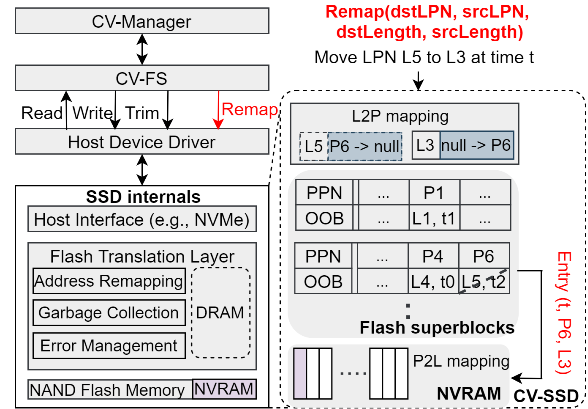


Figure 7: The REMAP command workflow for capacity variance: data in the range between `srcLPN` and `srcLPN + srcLength -1` are remapped to logical address starting from `dstLPN`. The third argument, `dstLength`, is optionally used for the file system to ensure I/O alignment.

3.1.3 Interface Changes for Capacity Variance

To integrate the address remapping approach into CV-FS, we revise the interface proposed by prior works [49, 68], `REMAP(dstLPN, srcLPN, dstLength, srcLength)`, and tailor it for capacity-variant storage systems. Our modified command enables file systems to safely shrink the logical capacity with minimal overhead by remapping valid data from their old LPNs to new LPNs without the need for actual data rewriting [49, 68]. We extend the current NVMe interface to include remap as a vendor-unique command.

Figure 7 shows an example of the remap command used for shrinking capacity. Assuming the file system address space ranges from LBA0 – LBA47 at the beginning (i.e., LPN0 – LPN5 with 512 bytes sector and 4 KiB page size) and LPN5 is mapped to PPN6 within the device. At time *t*, CV-FS initiates the capacity reduction and identifies that LBA40 – LBA47 (or LPN5) contains valid data. It then issues the remap command to move LPN5 data to LPN3 (i.e., `remap(LPN3, LPN5, 1, 1)`. Upon receiving the remap command, the FTL first finds the PPN associated with LPN5 (PPN6 in our case) and updates the logical-to-physical (L2P) mapping of L3 to PPN6. Finally, the old L2P mapping of L5 is invalidated, and the new physical-to-logical (P2L) mapping of PPN6 is recorded in the NVRAM of the SSD. Once the to-be-shrunk space is free, the file system states are validated and a new logical capacity size is updated.

In particular, the required size for NVRAM is small (for example, 1 MiB for a 1 TiB drive as suggested by the prior work [49]), as it is only used to maintain a log of the remapping metadata. Assuming the SSD capacity and page size are 1TB and 4KB, respectively, a single remapping entry requires

no more than 8B space. The 1 MiB NVRAM would be sufficient to hold entries for 512 MiB remapped data during a capacity reduction event. Between capacity reduction events, the reclamation of a flash block/page will cause a passive recycle on the associated remapping entries. However, in cases where a larger buffer is needed, the log can perform an active cleaning or write part of the mappings to flash because the space allocated for internal metadata will be conserved with a smaller device capacity [17]. Alternatively, the need for NVRAM can be eliminated by switching to the data relocation method, but at the cost of a higher overhead for logical capacity adjustment.

As a result, this new interface does not require taking the file system and device offline to adjust address space since data are managed logically. Moreover, it does not complicate the existing file system consistency management scheme. Similar to other events such as `discard`, the file system periodically performs checkpoints to provide a consistent state. The crash consistency is examined by manually crashing the system after initiating the remapping command and CrashMonkey [46] with its pre-defined workloads [45].

3.2 Capacity-Variant SSD

In this section, we outline design decisions and their leading benefits for building a capacity-variant SSD. We first discuss the necessity to forgo wear leveling in CV-SSD, and then describe its block management and life cycle management for extending lifetime and maintaining performance. Lastly, we introduce a degraded mode to handle the case where the remaining physical capacity becomes low.

Note that blocks in this subsection refer to physical flash memory blocks, different from the logical blocks managed by the file system. Furthermore, the flash memory blocks are grouped and managed as superblocks (again, different from the file system's superblock) to exploit the SSD's parallelism.

3.2.1 Wear Focusing

The goal of a capacity-variant SSD is to keep as much flash as possible at peak performance and mitigate the impact of underperforming and aged blocks. A capacity-variant SSD would maintain both performance and reliability by gracefully reducing its exported capacity so aged blocks can be mapped out earlier. Therefore, a capacity-variant SSD does not perform wear leveling (WL), as it degrades all of the blocks over time. WL is an artifact designed to maintain an illusion of a fixed-capacity device wherein its underlying storage components (i.e., flash memory blocks) either all work or fail, opposing our goal of allowing partial failure.

Moreover, static WL [8, 9, 15] incurs additional write amplification due to data relocation within an SSD. Dynamic WL [10, 11], on the other hand, typically combines with SSD internal tasks such as garbage collection, reducing the overall

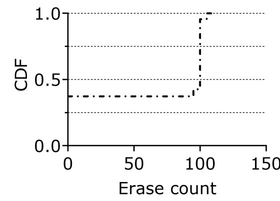


Figure 8: The wear distribution for a 256 GiB SSD under 100 iterations of MS-DTRS workload [29]. Traditional GC and block allocation policies cause a sudden capacity loss as too many blocks are equally aged.

cleaning efficiency as its victim selection considers both the valid ratio and wear state. A recent large-scale field study on millions of SSDs reveals that the WL techniques in modern SSDs present limited effectiveness [43] and an analysis study demonstrates that WL algorithms can even exhibit unintended behaviors by misjudging the lifetime of data in a block [25]. Such counter-productive results are avoided by forgoing WL and adopting capacity variance.

3.2.2 Block Management

A capacity-variant SSD exploits the characteristics of flash memory blocks to extend its lifetime and meet different performance and reliability requirements. Flash memory blocks in SSDs wear out at different rates and are marked as bad blocks by the bad block manager when they are no longer usable [26, 50]. This means that the physical capacity of the SSD naturally reduces over time, and for a fixed-capacity SSD, the entire storage device is considered to have reached the end of its life when the number of bad blocks exceeds its reserved space. On the other hand, the capacity-variant SSD's lifetime is defined by the amount of data stored in the SSD, rather than the initial logical capacity, making it a more reliable and efficient option.

The fail-slow symptoms and performance degradation in SSDs are caused by aged blocks with high error rates [4, 5, 42, 53]. Traditional SSDs consider blocks as either good or bad and such coarse-grained management fails to meet different performance and reliability requirements. On the other hand, the capacity-variant SSD defines three states of blocks: young, middle-aged, and retired, based on their operational characteristics. Young blocks have a relatively low erase count and a low RBER, while middle-aged blocks have higher errors and require advanced techniques to recover data. Retired blocks that are worn out or have a higher RBER than the configured threshold (5% by default) are excluded from storing data.

This block management scheme allows the capacity-variant SSD to map out underperforming and unreliable blocks earlier, effectively trading capacity for performance and reliability. In general, blocks start from a young state and transition to middle-aged and retired states. However, a block can also transition from a middle-aged state back to a young state since transient errors (i.e., retention and disturbance) are reset once the block is erased [30].

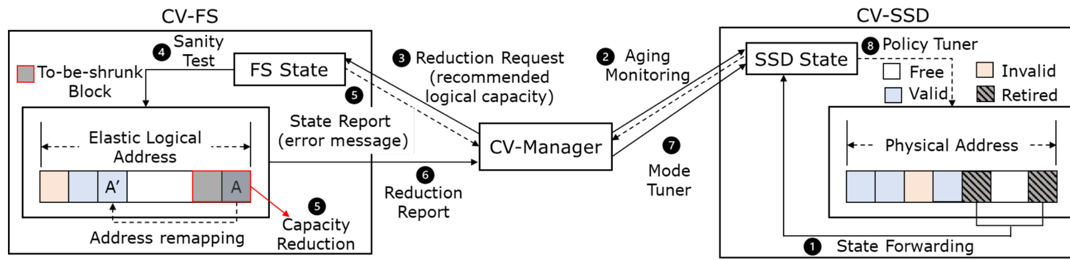


Figure 9: CV-manager design diagram. CV-manager monitors CV-SSD's aged state (Steps 1 and 2) and provides a recommended logical capacity to CV-FS (Step 3). After capacity reduction (Steps 4–6), CV-manager notifies CV-SSD (Step 7). The $CV_{degraded}$ mode will be triggered if the reduction fails (Step 8).

3.2.3 Life Cycle Management

A capacity-variant SSD requires wear focusing to mitigate the impact of aged flash memory blocks. However, simply avoiding wear leveling is insufficient as there are two processes affecting the life cycle of a flash memory block: **block allocation** and **garbage collection (GC)**. Traditional policies such as youngest-block-first for allocation and cost-benefit for GC work well on a traditional SSD, but are not suitable for CV-SSDs, since they aim to achieve a uniform wear state among blocks. Implementing these policies can cause a large number of blocks with the same erase count to map out simultaneously, leading to excessive capacity loss, and the device may suddenly fail. Figure 8 demonstrates this issue, where over 60% of the blocks aggregate to a particular wear state. Excessive capacity loss can increase the write amplification factor (WAF), particularly when the device utilization rate is high.

Allocation policy. In order to make wear accumulate in a small subset of blocks and allow capacity to shrink gradually, CV-SSD will prioritize middle-aged blocks to accommodate host writes and young blocks for GC writes. Since retired blocks are not used, there are four scenarios when considering data characteristics.

I. Write-intensive data are written to a middle-aged block

II. Write-intensive data are written to a young block

III. Read-intensive data are written to a middle-aged block

IV. Read-intensive data are written to a young block

Type I and type IV are ideal cases as they help to converge the wear among blocks without affecting the performance. Type II will also not affect the performance when data are fetched by the host because of the low RBER of young blocks. Moreover, with CV-SSD's allocation policy, such write-intensive data are inevitably re-written by the file system to the middle-aged blocks and the type II blocks will be GC-ed due to their low valid ratio. This type of scenario also happens under the early stages of CV-SSD, in which most blocks are young. Lastly, type III is the case where we need to pay more attention: read-intensive data should be stored in young blocks; otherwise expensive error correction techniques are triggered more often.

Garbage collection. We modify the garbage collection policy to consider (in)valid ratio, aging status, and data characteristics to handle type III cases. The block with the highest score will be selected as the victim based on the following formula:

$$\begin{aligned}
Victim\ score &= W_{invalidity} \cdot invalid\ ratio \\
&\quad + W_{aging} \cdot aging\ ratio \\
&\quad + W_{read} \cdot read\ ratio \\
invalid\ ratio &= \frac{\#\ of\ invalid\ pages}{\#\ of\ valid\ pages + \#\ of\ invalid\ pages}, \\
aging\ ratio &= \frac{erase\ count}{endurance}, \\
read\ ratio &= \frac{\#\ of\ host\ read\ designated\ to\ the\ current\ block}{maximum\ host\ read\ among\ unretired\ blocks}.
\end{aligned} \tag{1}$$

$W_{invalidity}$, W_{aging}, and W_{read} are weights to balance WAF, the aggressiveness of wear focusing, and the sensitivity of preventing type III scenarios, respectively. With that, read-intensive data stored in aged blocks are relocated by GC. Considering the read ratio could potentially affect the garbage collection efficiency. To avoid low GC efficiency, we set $W_{invalidity} = 0.4$, $W_{aging} = 0.3$, and $W_{read} = 0.3$, and their sensitivity analysis is shown in § 5.4.3. Increasing W_{read} is unfavorable not only because of adverse effects on WAF but also due to introducing unnecessary data movement. For example, a middle-aged block containing many valid pages but experiencing only a minimal number of reads is selected as the victim.

3.2.4 Degraded Mode

During normal conditions, CV-SSD intentionally unevens the wear state among blocks. As error-prone blocks retire, the physical capacity decreases gradually and performance is maintained. However, the physical capacity could decrease to a level where it will be insufficient to maintain current user data. Moreover, it can also cause high garbage collection overhead. In this case, $CV_{degraded}$ mode will be triggered and CV-SSD will slow down the further capacity loss. It is noteworthy that the triggering of degraded mode indicates a low remaining capacity to trade for performance, and storage administrators can gradually upgrade storage systems.

In particular, the $CV_{degraded}$ mode is triggered under two conditions: (1) when the effective over-provisioning

(EOP), calculated as $EOP = (physical\ capacity - utilization)/utilization$, falls below the factory-set over-provisioning (OP), or (2) when the remaining physical capacity is less than a user-defined watermark.

Once $CV_{degraded}$ mode is set, GC only considers WAF and aging to slow down further capacity loss. This mode allows young blocks to be cleaned with a relatively higher valid ratio than aged blocks. Specifically, young blocks with a high invalid ratio are optimal candidates. Moreover, middle-aged blocks are used to accommodate GC-ed data, and young blocks are allocated for host writes. As a result, blocks are used more evenly than in the initial stage and a particular amount of physical capacity is maintained for the user. When EOP becomes greater than OP if the host decides to move or delete some data, $CV_{degraded}$ will be reset by CV-manager.

3.3 Capacity-Variant Manager

To improve usability, CV-manager is responsible for automatically managing the capacity of the whole storage system. As illustrated in Figure 9, CV-manager monitors the aged state of the underlying storage device and provides a recommended logical partition size to the kernel.

Specifically, when CV-SSD maps out blocks and its physical capacity is reduced, CV-manager will get notified (Steps 1 and 2). The CV-manager figures out a recommended logical capacity by checking the current bad capacity within the device and issues capacity reduction requests to CV-FS through a system call (Step 3). Upon request, CV-FS performs a sanity test. If the file system checkpoint functionality is disabled or the file system is not ready to shrink (i.e., frozen or read-only), the reduction will not continue (Step 4). Otherwise, CV-FS starts shrinking capacity as described in § 3.1.3 and returns the execution result (Steps 5 and 6). Lastly, CV-manager notifies CV-SSD whether logical capacity is reduced properly or not. If the reduction fails, the $CV_{degraded}$ is activated to slow down further capacity loss (Steps 7 and 8).

For user-level capacity management, CV-manager provides necessary interfaces for users to explicitly initiate capacity reduction and set performance and reliability requirements for the device. The CV-SSD would retire blocks based on the host requirement. Similar to the read recovery level (RRL) command [47] in the NVMe specification that limits the number of read retry operations for a read request, this configurable attribute limits the maximum amount of recovery applied to a request and thus balances the performance.

4 Implementation

The capacity-variant file system (CV-FS) is implemented upon the Linux kernel v5.15. CV-FS uses F2FS [35] as the baseline file system due to its virtue of being a log-structure file system. We modify both CVSS and TrSS to employ a more aggressive discard policy than the baseline F2FS (i.e., 50ms interval if candidates exist and 10s max interval if no candidates) for better SSD garbage collection efficiency [32] (also shown in § 5.2.3).

To implement the `remap` command, we extend the block I/O layer. A new I/O request operation `REQ_OP_REMAP` is added to expose the remap command to the CV-FS. New attributes including `bio->bi_iter.bi_source_sector` and `bio->bi_iter.bi_source_size` are introduced in `bvec_iter`, which corresponds to the second and last parameter of the remap command. Functions related to `bio` splitting/merging procedure (e.g., `__blk_queue_split`) are modified to maintain added attributes (mainly in `/block/blk-merge.c`). Additionally, new `nvme_opcode` and related functions are added to support the remap command at the NVMe driver layer (mainly in `/block/blk-mq.c` and `/drivers/nvme/host/core.c`).

The capacity-variant SSD is built on top of the FEMU [37]. SSD reliability enhancement techniques such as ECC and read retry ensure data integrity. To implement the error model, we use the additive power-law model proposed in prior works [30, 39, 44] that considers wear, retention loss, and disturbance to quantify RBER, as shown in the following equation:

$$\begin{aligned} &RBER(cycles, time, reads) \\ &\quad = \varepsilon + \alpha \cdot cycles^k && (wear) \\ &\qquad + \beta \cdot cycles^m \cdot time^n && (retention) \\ &\qquad + \gamma \cdot cycles^p \cdot reads^q && (disturbance) \end{aligned} \tag{2}$$

The parameters used are particular to a real 2018 TLC flash chip [30], and the device internally keeps track of cycles, time, and reads for each block. During a read operation, read retry is applied if the error exceeds the ECC strength. We consider each read retry will lower the error rate by half [30, 53] and the maximum amount of recovery is limited for a single read retry so that blocks have more fine-grained error states.

We modify five major software components to support capacity variance.

- We make changes to the Linux kernel v5.15 to provide an `ioctl`-based user-space API supporting logical partition reduction. Users can specify the shrinking size and issue capacity reduction commands through this API.
- We modify the F2FS to handle address remapping triggered by capacity variance and revise its discard scheme.
- We extend the `f2fs-tool` (f2fs format utility) to support the CV-specific functionalities, such as initializing a variable logical partition and updating the attributes that control discard policies.
- We implement CV-SSD mode in FEMU, adding flash reliability enhancement techniques, error models, wear leveling, bad block management, and device lifetime features.
- We modify NVMe device driver and introduce new commands to `NVMe-Cli` [48], to support capacity variance. The

Table 1: System configurations. Wear leveling (PWL [9]) and youngest block first allocation are used for traditional SSDs.

PC platform			
Parameter	Value	Parameter	Value
CPU name	Intel Xeon 4208	Frequency	2.10GHz
Number of cores	32	Memory	1TiB
Kernel	Ubuntu v5.15	ISA	X86_64
FEMU			
Parameter	Value	Parameter	Value
Channels	8	Physical capacity	128 GiB
Luns per channel	8	Logical capacity	120 GiB
Planes per lun	1	Over-provisioning	7.37%
Blocks per plane	512	Garbage collection	Greedy
Pages per block	1024	Program latency	500 μs
Page size	4 KiB	Read latency	50 μs
Superblock size	256 MiB	Erase latency	5 ms
Endurance	300	Wear leveling	PWL [9]
ECC strength	50 bits	Block allocation	Youngest first

SMART [47] command is also extended to export more device statistics for capacity management.

5 Evaluation of Capacity Variance

We first describe our experimental setup and methodology, then present our evaluation results and demonstrate the effectiveness of capacity variance.

5.1 Experimental Setup and Methodology

Table 1 outlines the system configurations for our evaluation. For the traditional SSD, an adaptive WL, PWL [9], is used to even the wear among blocks. The error correction code (ECC) for both Tr-SSD and CV-SSD is configured to tolerate up to 50-bit errors per 4 KiB data, and errors beyond the correction strength are subsequently handled by read retry. We use 17 different workloads in our evaluation: (1) 4 FIO [23] workloads (Zipfian and random, each with two different utilization); (2) 3 Filebench [60] workloads; (3) 2 YCSB workloads [2] (YCSB-A and YCSB-F); and (4) 8 key-value traces from Twitter [64].

We compare CVSS with three different techniques: (1) TrSS, a traditional storage system with vanilla F2FS plus a fixed-capacity SSD; (2) AutoStream [63]; (3) ttFlash [62]. The evaluation comparisons are selected based on their broader applicability and implementation simplicity of the multi-stream interface (represented by AutoStream [63]) and the fast-fail mechanism (represented by ttFlash [62]). These approaches align with more general and widely used methods such as PCStream [33], LinnOS [22], and IODA [38]. Specifically, AutoStream [63] uses the multi-stream interface [28] and automatically assigns a stream ID to the data based on the I/O access pattern. The SSD then places data accordingly based on the assigned ID to reduce write amplification and thus, improve performance. On the other hand, ttFlash [62] reduces the tail latency of SSDs by utilizing a redundancy scheme (similar to RAID) to reconstruct data when blocked by GC. Since the original ttFlash is implemented on a simulator, we implement its logic in FEMU for a fair comparison.

To perform a more realistic evaluation, it is necessary to reach an aged FS and device state. Issuing workloads manually to the system is prohibitively expensive, as it takes years' worth of time. Moreover, this method lacks standardization and reproducibility, making the evaluation ineffective [1]. We extensively use aging frameworks in our evaluation. Prior to each experiment, we use impression [1] to generate a representative aged file system layout. After file system aging, the fast-forwardable SSD aging framework (FF-SSD) [26] is used to reach different aged states for SSD. The aging acceleration factor (AF) is strictly limited to 2 to maintain accuracy. Workloads will run until the underlying SSD fails.

We design the experiments with the following questions:

- Can CVSS maintain performance while the underlying storage device ages? (§ 5.2)
- Can CVSS extend the device lifetime under different performance requirements? (§ 5.3)
- What are the tradeoffs in CVSS design? (§ 5.4)

5.2 Performance Improvement

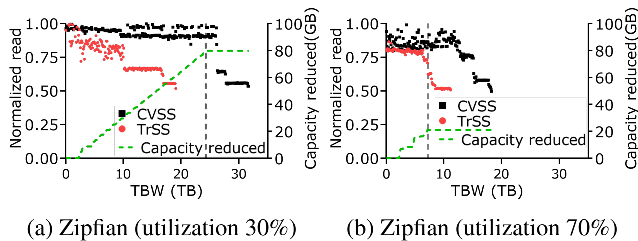
In this section, we evaluate the effectiveness of CVSS in maintaining performance and avoiding fail-slow symptoms under synthetic and real workloads.

5.2.1 FIO

We first examine the performance benefit of capacity variance under Zipfian workloads with two different workload sizes: 38GB (utilization of 30%) and 90GB (utilization of 70%). For this experiment, FIO continuously issues 16KB read and write requests to the device. We use the default setting of FIO and the read/write ratio is 0.5/0.5.

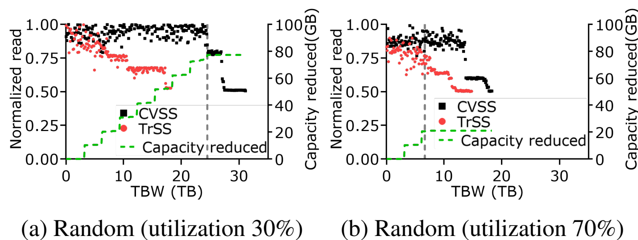
Zipfian. Figure 10 shows the read throughput under different aged states of TrSS and CVSS, in terms of terabytes written (TBW). We measure the performance until it drops below 50% of the initial value where no aging-related operations are performed. The green dotted line shows the amount of physical capacity that has been reduced within the CV-SSD and the straight vertical line represents the trigger of $CV_{degraded}$.

We observe that TrSS and CVSS behave similarly at first where both CV-SSD and Tr-SSD are relatively young. However, for TrSS, the read performance degrades gradually. As Tr-SSD gets aged, the amount of error corrected during each read operation increases and thus involves more expensive read retry processes. On the other hand, CV-SSD effectively trades the capacity for performance. The performance is maintained by excluding heavily aged blocks from use. Later, $CV_{degraded}$ is triggered to maintain a particular amount of capacity for the workloads. During this stage, blocks are used



(a) Zipfian (utilization 30%) (b) Zipfian (utilization 70%)

Figure 10: Read throughput under FIO Zipfian workloads. In CVSS, the performance is maintained by trading capacity. The straight vertical line represents the trigger of the $CV_{degraded}$ mode. After $CV_{degraded}$, the future capacity reduction is slowed down but the performance is compromised.



(a) Random (utilization 30%) (b) Random (utilization 70%)

Figure 11: Read throughput under FIO random workloads. CVSS delivers up to 0.6× (left) and 0.7× (right) higher performance compared to TrSS, under the same amount of host writes.

more evenly and the wear accumulates within the device. In this case, performance is traded for capacity in order to avoid data loss. However, even in this mode, CVSS delivers better performance compared to TrSS, thanks to the previous mapping out of most unreliable blocks. Overall, the read throughput of CVSS outperforms TrSS by up to 0.72× with the same amount of host writes.

Random. Figure 11 shows the measured read performance under random I/O. The configuration is similar to the previous case. As in-used blocks get aged, the read performance of TrSS degrades gradually and the fail-slow symptoms manifest. With the same amount of host writes, CVSS delivers a 0.6× and a 0.7× higher throughput at most than TrSS under the utilization of 30% and 70%, respectively.

Figure 13 compares the average write performance over the measurement. For Tr-SSD, when WL is initiated, data are relocated within the device, which decreases the throughput by 0.6×. Without WL, CV-SSD provides a more stable and better write performance than Tr-SSD. Overall, the write throughput of CVSS outperforms TrSS by 0.12× on average.

5.2.2 Filebench

We now use Filebench [60] to evaluate the capacity-variant system under file system metadata-heavy workloads. We use three pre-defined workloads in the benchmark, which exhibit differences in I/O patterns and fsync usage.

Figures 12a, 12b, and 12c show the CDF of operation latency under fileserver, netsfs, and varmail workloads throughout the devices' life. In particular, CVSS–normal represents the result before $CV_{degraded}$ is activated and CVSS represents the overall result. We use the default setting of Filebench, which measures the performance by running workloads for 60 seconds. Random writes are used to age CV-SSD and Tr-SSD. The measurement is performed after every 100GB of random data written until the device fails. The utilization for both TrSS and CVSS is 50%.

Compared to TrSS, CVSS reduces the average response time by 32% before the degraded mode is triggered and by 24% over the entire lifetime under netsfs workload, as shown in Figure 12b. The netsfs workload simulates the behavior of a network file server. It performs a more comprehensive set of operations such as application lunch, read-modify-write, file appending, and metadata retrieving, and thus reflects the state of the underlying devices more intuitively. Overall, CVSS reduces the average latency by 8% in the fileserver case (Figure 12a), and 10% in the varmail case (Figure 12c).

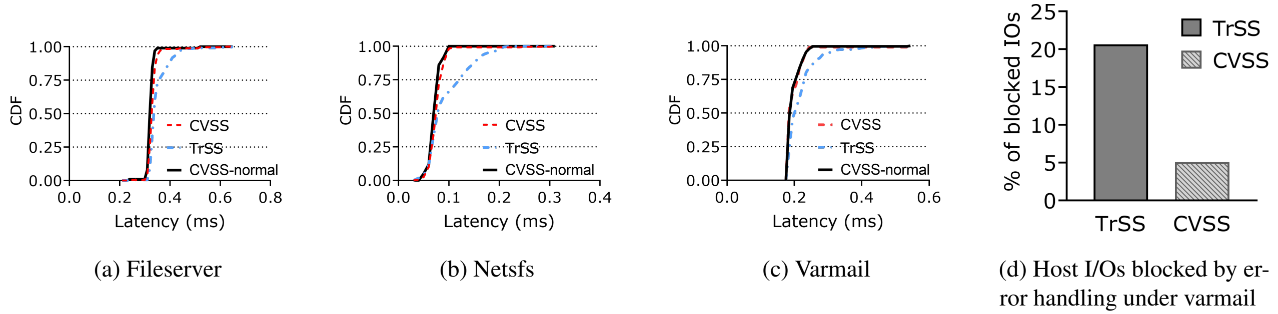
CV-SSD maps out blocks once their RBER exceeds 5% by default, while Tr-SSD only maps them out when their erase counts exceed the endurance limit, leading to more expensive error correction operations. The increased error correction operations not only affect the latency of the ongoing host request but also create backlogs in IO traffic. Figure 12d shows the percentage of host I/Os blocked by read retry operations measured inside FEMU under the varmail workload. In TrSS, more than 20% of I/O requests are delayed by SSD internal read retry, while it is no more than 5% in CVSS.

5.2.3 Twitter Traces

The previous sections examine CVSS using block I/O workloads and file system metadata-heavy workloads. In this section, we evaluate CVSS and compare it with AutoStream [63] and ttFlash [62] at the overall application level. We use a set of key-value traces from Twitter production [64]. The Twitter workload contains 36.7 GB worth of key-value pairs in total. We first load the key-values pairs and then start and keep feeding the traces to RocksDB until the underlying SSD fails.

Figure 14 compares the average KIOPS over the entire device lifetime. Overall, capacity variance improves the throughput by 0.49× – 3.16× compared to TrSS; on the other hand, AutoStream and ttFlash present limited effectiveness in mitigating fail-slow symptoms. In particular, Trace38 highlights the benefits of capacity variance, achieving a 3.16× better throughput than the fixed-capacity storage. For RocksDB, point lookups may end up consulting all files in level 0 and at most one file from each of the other levels. Therefore, as the Tr-SSD ages, a single `Get()` request can cause multiple physical reads and each of them can trigger SSD read retry several times, degrading the read performance drastically.

Moreover, we find that traditional systems with the original discard policy show higher utilization inconsistency (i.e.,



(a) Fileserver

(b) Netsfs

(c) Varmail

(d) Host I/Os blocked by error handling under varmail

Figure 12: Performance results under Filebench workloads. CVSS reduces the average latency by 8% under fileserver workload (Figure 12a), 24% under netsfs workload (Figure 12b), and 10% under varmail workload (Figure 12c) compared to TrSS throughout the devices' lifetime. Before $CV_{degraded}$ is triggered, CVSS–normal reduces the average latency by 32% under netsfs workload. Figure 12d shows the percentage of host I/Os blocked by read retry operations under varmail workload. Other workloads show a similar pattern.

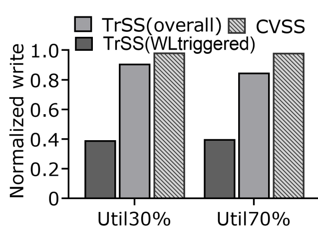


Figure 13: Average write throughput under FIO workloads. For TrSS, when wear leveling is triggered, the write throughput drops by 0.6×; on the other hand, by forgoing WL, CV-SSD provides a more stable and better write performance.

$\frac{1}{n}\sum_{Observation=1}^{n} util_{SSD} - util_{FS})$ between FS and SSD, as shown in Figure 16. That is because of the high request rate during the experiments and F2FS only dispatches discard command when the device I/O is idle, which not only decreases SSD GC efficiency but also makes wear leveling more likely to misjudge data aliveness, limiting its effectiveness in maintaining capacity. During the experiments, the WAF of TrSS can be as high as 6.79, while only 1.12 for CVSS.

5.3 Lifetime Extension

In this section, we investigate how CVSS extends device lifetime given different performance requirements and thus leads to a longer replacement interval for SSD-based storage systems. We compare three different configurations: CVSS, TrSS, and AutoStream [63] in this evaluation since ttFlash introduces additional write (wear) overhead coming from RAIN (Redundant Array of Independent NAND) even for a small write [62]. The workloads used are similar to § 5.2.1.

Figure 15 shows the TBW before the device performance drops below 0.8, 0.6, 0.4, and 0 of the initial state. In particular, 0 represents the case where no performance requirement is applied so the workload runs until the underlying SSD is unusable. In cases of lower device utilization (as shown in Figures 15a and 15c), CVSS effectively extends the device lifetime, even when high performance is required. In Figure 15a, the device fails after accommodating 10 TB host writes for TrSS and 18 TB for AutoStream, considering the performance requirement of 0.8. On the other hand, CVSS accommodates 28 TB host writes with the same performance

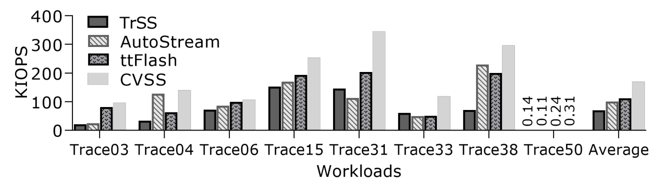


Figure 14: Performance results under Twitter traces. Capacity variance outperforms AutoStream and ttFlash and improves the throughput by 1.42× on average compared to TrSS.

requirement, outlasting TrSS by 180% and AutoStream by 55%. Similarly, in Figure 15c, CVSS outlasts TrSS by 270% and AutoStream by 50%.

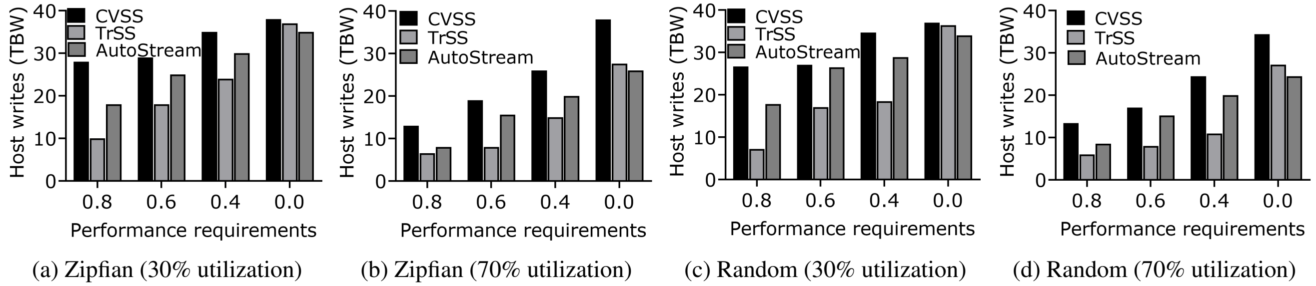
In the high device utilization cases (as shown in Figure 15b and 15d), CVSS outlasts TrSS by 123% and AutoStream by 55% on average with the highest performance requirement. In Figure 15b, before the device becomes unusable, CVSS accommodates 10.4 TB more in host writes compared to TrSS and 12 TB more compared to AutoStream. In our experiments, we found AutoStream achieves a longer lifetime than TrSS except for the no performance requirement case. In AutoStream, data are placed based on their characteristics, which in turn triggers more data relocation towards the end for wear leveling. Overall, with the highest performance requirement, CVSS ingests 168% host data more compared to TrSS and 57% more compared to AutoSteam on average, which in turn prolongs the replacement interval and reduces the cost.

5.4 Sensitivity Analysis

We next investigate the tradeoffs in CVSS regarding the block retirement threshold, the strength of ECC engine, and the impact of different GC formula weights.

5.4.1 Block Retirement Threshold

The mapping-out behavior for aged blocks in CV-SSD is controlled by a user-defined threshold. By default, blocks



(a) Zipfian (30% utilization) (b) Zipfian (70% utilization) (c) Random (30% utilization) (d) Random (70% utilization)

Figure 15: Terabytes written (TBW) with different performance requirements. Compared to TrSS and AutoStream, CVSS significantly extends the lifetime while meeting performance requirements.

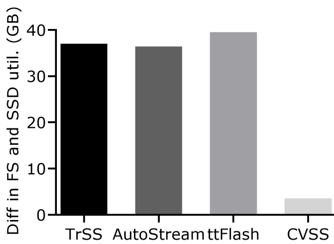


Figure 16: The average difference in FS and SSD utilization under Twitter traces. The original discard policy shows higher utilization inconsistency between FS and SSD, making data aliveness misjudged.

are mapped out and turn to a retired state once their RBER exceeds 5%. In this section, we investigate how this threshold affects the performance and device lifetime.

We utilize YCSB-A and YCSB-F with their data set configured to have thirty million key-value pairs (10 fields, 100 bytes each, plus key). We compare three different configurations: (1) TrSS, vanilla F2FS plus a fixed-capacity SSD; (2) CVSS(4%), CVSS with a higher reliability requirement. Superblocks will be mapped out if RBER is greater than 4%; (3) CVSS(6%), CVSS with a lower reliability requirement. Superblocks will be mapped out if RBER is greater than 6%.

Figure 17 shows the latencies at major percentile values (p75 to p99) and the device lifetimes for each workload. As shown in Figures 17a and 17b, CVSS(4%) reduces p99 latency by 51% for the YCSB-A workload and by 53% for the YCSB-F workload compared to TrSS, which are 44% and 40% for CVSS(6%). With a higher reliability requirement, blocks are retired earlier in CVSS(4%), which in turn causes a relatively shorter device lifetime than CVSS(6%). As depicted in Figures 17c and 17d, CVSS(6%) and CVSS(4%) ingests 3.27× and 2.68× host I/O than TrSS on average, respectively.

5.4.2 ECC Strength

We now study the impact of ECC strength on SSD error handling and demonstrate the usefulness of capacity variance in simplifying SSD FTL design. As discussed earlier, CVSS excludes aged blocks from use and thus incurs fewer error correction operations. This further allows the CV-SSD to be equipped with a less robust error-handling mechanism without compromising reliability.

Figure 18 compares the average number of read retries triggered per GiB read over the device's lifetime for CVSS with ECC strength set as up to 50 bits corrected per 4KiB and TrSS with ECC strength set to 50 – 90 bits. The results are measured under the FIO Zipfian read/write workload with device utilization of 30%. We make two observations. First, with the same ECC capability, TrSS(50) performs 1.93× more read retry operations than CVSS(50). Second, TrSS requires a stronger ECC engine to improve the efficiency of the error correction process, which complicates the FTL design in SSDs. On the other hand, with a weaker ECC engine, CVSS(50) achieves similar performance to TrSS(90).

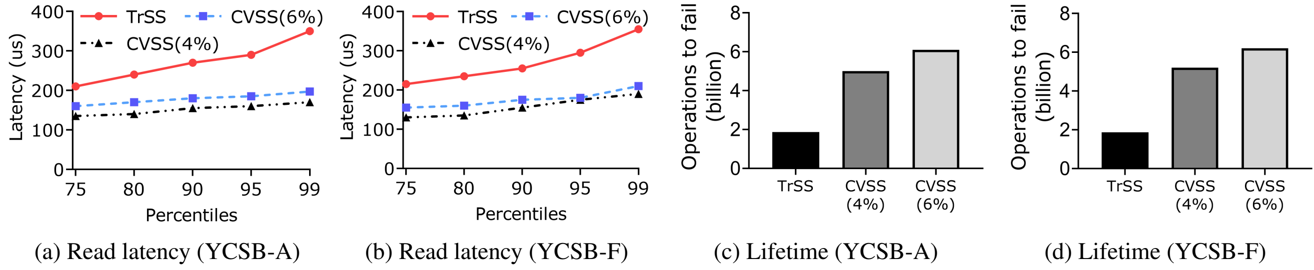
5.4.3 GC Formula

As described in § 3.2.3, the GC formula consists of three parameters: $W_{invalidity}$, W_{aging}, and W_{read}. We analyze how different weights used in GC formula affect the performance of CVSS in this section. We compare the configured weights with three different configurations: (1) GC prioritizes blocks with more invalid pages, with $W_{invalidity}$ = 0.6, W_{read} = 0.2, and W_{aging} = 0.2; (2) GC prioritizes blocks with more reads, with $W_{invalidity}$ = 0.2, W_{read} = 0.6, and W_{aging} = 0.2; (3) GC prioritizes blocks with more erases, with $W_{invalidity}$ = 0.2, W_{read} = 0.2, and W_{aging} = 0.6. FIO is used to generate Zipfian read-/write workloads to the device. Figure 19 illustrates the measured WAF and read retry. Overall, the configured weights result in a lower WAF and fewer read retry operations.

In particular, compared to the configured case, the high $W_{invalidity}$ case achieves a lower WAF but involves 0.78× more read retry operations. This is because read-intensive data are stored in aged blocks. For the high W_{read} case, it triggers fewer read retry operations but decreases the cleaning efficiency of GC since the invalidity is not adequately considered during the victim selection. In the high W_{aging} case, GC always selects the most aged blocks, leading to a significant increase in WAF and faster device aging. In contrast, the configured weights balance WAF and read retry within the device.

6 Discussion and Future Work

In this section, we discuss different use cases of capacity variance and its intersection with ZNS and RAID systems.



(a) Read latency (YCSB-A) (b) Read latency (YCSB-F) (c) Lifetime (YCSB-A) (d) Lifetime (YCSB-F)

Figure 17: Sensitivity analysis on the mapping-out threshold in CVSS. CVSS with a higher reliability requirement, CVSS(4%), achieves better performance but with a relatively shorter lifetime compared to CVSS(6%) because blocks are retired earlier.

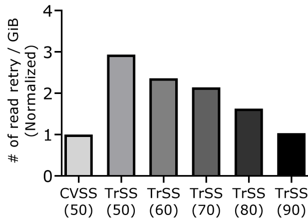


Figure 18: The average number of read retries triggered per GiB read over the device's lifetime. The x-axis represents different ECC strengths in bits.

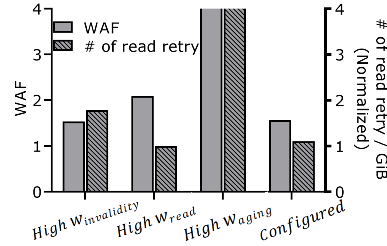


Figure 19: The WAF and read retries triggered under different weights used for GC formula.

Use cases of capacity variance. CVSS aims to significantly outperform fixed-capacity systems in the best case, and perform at a similar level in the worst case. The degraded mode serves the role of addressing the worst case by reserving a particular amount of capacity for the host. CVSS would be most useful for cases where IO performance is bottlenecked but has spare capacity. For instance, Haystack is the storage system specialized for new blobs (Binary Large Objects) and bottlenecks on IOPS but has spare capacity [51].

Moreover, for SSD vendors, capacity variance can simplify SSD design, as it allows for the tradeoff of performance and reliability with capacity. For data centers, introducing capacity variance can automatically exclude unreliable blocks and enable easy monitoring of device capacity, resulting in longer device replacement interval and mitigating SSD failure-related issues in data center environments. Lastly, for desktop users, capacity variance extends the lifetime of SSDs significantly and thus reduces the overall cost of storage.

ZNS-SSD. Capacity variance can be harmonious with ZNS. Specifically, due to a wear-out, a device may (1) choose to take a zone offline, or (2) report a new, smaller size for a zone after a reset. Both of these result in a shrinking capacity SSD. However, there is no software that can handle capacity reduction for ZNS-SSDs currently. The offline command simply makes a zone inaccessible and data relocation has to be done by users. Moreover, file systems are typically unaware of this change except for ZoneFS [34]. The capacity-variant SSD interface is a more streamlined solution where the software and the hardware cooperate to automate the process.

Capacity variance with RAID. The current CVSS does not support RAID systems. Existing RAID architectures require symmetrical capacity across devices and its overall capacity depends on the underlying minimal-capacity device. For parity RAID, the invalid data can not always be trimmed because it may be required to ensure the parity correctness. We will investigate the capacity-variant RAID system as our next direction, in which we consider modifying the disk layout and data placement scheme to support dynamically changing asymmetrical capacity with multiple heterogeneous CV-SSDs.

7 Conclusion

The basic principle behind a capacity-variant storage system is simple: relax the fixed-capacity abstraction of the underlying storage device. We implement this idea and describe the key designs and implementation details of a capacity-variant storage system. Our evaluation result demonstrates how capacity variance leads to performance advantages and shows its effectiveness and usefulness in avoiding SSD fail-slow symptoms and extending device lifetime. We expect new optimizations and features will be continuously added to the capacity-variant storage system.

Acknowledgment

We would like to thank our shepherd, Keith A. Smith, and the anonymous reviewers for their constructive feedback. This research was supported in part by the National Science Foundation (NSF CNS-2008453), Samsung Memory Solutions Lab through the Alternative Sustainable and Intelligent Computing Industry-University Cooperative Research Center (NSF CNS-1822165), and Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Ministry of Science and ICT (MSIT), Korea (No. 2021-0-01475, SW Starlab).

References

[1] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating realistic impressions for file-system benchmarking. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 125–138. USENIX, 2009.

[2] Brian Cooper. Yahoo Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB/, 2010.

[3] Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE*, pages 1666–1704, 2017.

[4] Yu Cai, Erich F. Haratsch, Onur Mutlu, and Ken Mai. Threshold voltage distribution in MLC NAND flash memory: characterization, analysis, and modeling. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1285–1290. ACM, 2013.

[5] Yu Cai, Yixin Luo, Erich F. Haratsch, Ken Mai, and Onur Mutlu. Data retention in MLC NAND flash memory: Characterization, optimization, and recovery. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 551–563. IEEE Computer Society, 2015.

[6] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F. Haratsch, Adrián Cristal, Osman S. Ünsal, and Ken Mai. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *International Conference on Computer Design (ICCD)*, pages 94–101. IEEE Computer Society, 2012.

[7] Chandranil Chakraborttii and Heiner Litz. Improving the accuracy, adaptability, and interpretability of SSD failure prediction models. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SOCC)*, pages 120–133. ACM, 2020.

[8] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *ACM Symposium on Applied Computing (SAC)*, pages 1126–1130, 2007.

[9] Fu-Hsin Chen, Ming-Chang Yang, Yuan-Hao Chang, and Tei-Wei Kuo. PWL: a progressive wear leveling to minimize data migration overheads for NAND flash devices. In *Design, Automation & Test in Europe Conference & Exhibition, (DATE)*, pages 1209–1212, 2015.

[10] Zhe Chen and Yuelong Zhao. DA-GC: A dynamic adjustment garbage collection method considering wear-leveling for SSD. In *Great Lakes Symposium on VLSI (GLSVLSI)*, pages 475–480, 2020.

[11] Mei-Ling Chiang, Paul CH Lee, and Ruei-Chuan Chang. Using data clustering to improve cleaning performance for flash memory. *Software: Practice and Experience*, pages 267–290, 1999.

[12] Alex Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, and Martin Farach-Colton. File systems fated for senescence? nonsense, says science! In *USENIX Conference on File and Storage Technologies (FAST)*, pages 45–58. USENIX, 2017.

[13] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A. Bender, William Jannen, Rob Johnson, Donald E. Porter, and Martin Farach-Colton. Filesystem aging: It's more usage than fullness. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, pages 15–21. USENIX, 2019.

[14] Yajuan Du, Siyi Huang, Yao Zhou, and Qiao Li. Towards LDPC read performance of 3D flash memories with layer-induced error characteristics. *ACM Transactions on Design Automation of Electronic Systems*, pages 44:1–44:25, 2023.

[15] Thomas Gleixner, Frank Haverkamp, and Artem Bityutskiy. UBI - Unsorted Block Images. http://linux-mtd.infradead.org/doc/ubidesign/ubidesign.pdf, 2006.

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

[17] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 91–103. USENIX, 2011.

[18] Keonsoo Ha, Jaeyong Jeong, and Jihong Kim. An integrated approach for managing read disturbs in high-density NAND flash memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pages 1079–1091, 2016.

[19] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving file system performance of

mobile storage systems using a decoupled defragmenter. In *USENIX Annual Technical Conference (ATC)*, pages 759–771. USENIX, 2017.

[20] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting millisecond tail tolerance with fast rejecting slo-aware OS interface. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 168–183. ACM, 2017.

[21] Mingzhe Hao, Gokul Soundararajan, Deepak R. 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 *USENIX Conference on File and Storage Technologies (FAST)*, pages 263–276. USENIX, 2016.

[22] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on unpredictable flash storage with a light neural network. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–190. USENIX, 2020.

[23] Jens Axboe. Flexible I/O tester. https://github.com/axboe/fio/, 2005.

[24] Cheng Ji, Li-Pin Chang, Sangwook Shane Hahn, Sungjin Lee, Riwei Pan, Liang Shi, Jihong Kim, and Chun Jason Xue. File fragmentation in mobile devices: Measurement, evaluation, and treatment. *IEEE Transactions on Mobile Computing (TMC)*, pages 2062–2076, 2019.

[25] Ziyang Jiao, Janki Bhimani, and Bryan S. Kim. Wear leveling in SSDs considered harmful. In *ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*, pages 72–78. ACM, 2022.

[26] Ziyang Jiao and Bryan S. Kim. Generating realistic wear distributions for SSDs. In *ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*, pages 65–71. ACM, 2022.

[27] Myoungsoo Jung and Mahmut T. Kandemir. Revisiting widely held SSD expectations and rethinking system-level implications. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 203–216. ACM, 2013.

[28] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, pages 13–17. USENIX, 2014.

[29] Swaroop Kavalanekar, Bruce L. Worthington, Qi Zhang, and Vishal Sharda. Characterization of storage workload traces from production Windows servers. In *International Symposium on Workload Characterization (IISWC)*, pages 119–128, 2008.

[30] Bryan S. Kim, Jongmoo Choi, and Sang Lyul Min. Design tradeoffs for SSD reliability. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 281–294. USENIX, 2019.

[31] Bryan S. Kim, Eunji Lee, Sungjin Lee, and Sang Lyul Min. CPR for SSDs. In *ACM Workshop on Hot Topics in Operating Systems (HotOS)*, pages 201–208. ACM, 2019.

[32] Juwon Kim, Minsu Kim, Muhammad Danish Tehseen, Joontaek Oh, and Youjip Won. IPLFS: log-structured file system without garbage collection. In *USENIX Annual Technical Conference (ATC)*, pages 739–754. USENIX, 2022.

[33] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Joo Young Hwang, Jongyoul Lee, and Jihong Kim. Fully automatic stream management for multi-streamed SSDs using program contexts. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 295–308. USENIX, 2019.

[34] Damien Le Moal and Ting Yao. Zonefs: Mapping POSIX file system interface to raw zoned block device accesses. In *Linux Storage and Filesystems Conference (VAULT)*, page 19. USENIX, 2020.

[35] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2FS: a new file system for flash storage. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 273–286. USENIX, 2015.

[36] Jaeyong Lee, Myungsuk Kim, Wonil Choi, Sanggu Lee, and Jihong Kim. Tailcut: improving performance and lifetime of SSDs using pattern-aware state encoding. In *ACM/IEEE Design Automation Conference (DAC)*, pages 409–414. ACM, 2022.

[37] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminatahan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *USENIX Conference on File and Storage Technologies (FAST)*, page 83–90. USENIX, 2018.

[38] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. IODA: A host/device co-design for strong predictability contract on modern flash storage. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 263–279. ACM, 2021.

[39] Ren-Shuo Liu, Chia-Lin Yang, and Wei Wu. Optimizing NAND flash-based SSDs via retention relaxation. In *USENIX Conference on File and Storage Technologies (FAST)*, page 11. USENIX, 2012.

[40] Ruiming Lu, Erci Xu, Yiming Zhang, Fengyi Zhu, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Jiwu Shu, Minglu Li, and Jiesheng Wu. Perseus: A fail-slow detection framework for cloud storage systems. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 49–64. USENIX, 2023.

[41] Ruiming Lu, Erci Xu, Yiming Zhang, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Minglu Li, and Jiesheng Wu. NVMe SSD failures in the field: the fail-stop and the fail-slow. In *USENIX Annual Technical Conference (ATC)*, pages 1005–1020. USENIX, 2022.

[42] Yixin Luo, Saugata Ghose, Yu Cai, Erich F. Haratsch, and Onur Mutlu. HeatWatch: Improving 3D NAND flash memory device reliability by exploiting self-recovery and temperature awareness. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 504–517. IEEE Computer Society, 2018.

[43] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. Operational characteristics of SSDs in enterprise storage systems: A large-scale field study. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 165–180, 2022.

[44] Neal R. Mielke, Robert E. Frickey, Ivan Kalastirsky, Minyan Quan, Dmitry Ustinov, and Venkatesh J. Vasudevan. Reliability of solid-state drives based on NAND flash memory. *Proceedings of the IEEE (JPROC)*, pages 1725–1750, 2017.

[45] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. CrashMonkey. https://github.com/utsaslab/crashmonkey/tree/master/code/tests/seq1, 2018.

[46] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 33–50. USENIX, 2018.

[47] NVM Express. NVM Express base specification 2.0. https://nvmexpress.org/developers/nvme-specification/, 2021.

[48] NVM Express. NVMe Command Line Interface. https://github.com/linux-nvme/nvme-cli, 2021.

[49] Gihwan Oh, Chiyoung Seo, Ravi Mayuram, Yang-Suk Kee, and Sang-Won Lee. SHARE interface in flash storage for relational and NoSQL databases. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, pages 343–354. ACM, 2016.

[50] Open NAND Flash Interface. ONFI 5.0 spec. http://www.onfi.org/specifications/, 2021.

[51] Satadru Pan, Theano Stavrinos, 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, J. R. Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's Tectonic filesystem: Efficiency from exascale. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 217–231. USENIX, 2021.

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

[53] Jisung Park, Myungsuk Kim, Myoungjun Chun, Lois Orosa, Jihong Kim, and Onur Mutlu. Reducing solid-state drive read latency by optimizing read-retry. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 702–716. ACM, 2021.

[54] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15. ACM, 1991.

[55] Xin Shi, Fei Wu, Shunzhuo Wang, Changsheng Xie, and Zhonghai Lu. Program error rate-based wear leveling for NAND flash memory. In *Design, Automation & Test in Europe Conference & Exhibition, (DATE)*, pages 1241–1246. IEEE, 2018.

[56] Youngseop Shim, Myungsuk Kim, Myoungjun Chun, Jisung Park, Yoona Kim, and Jihong Kim. Exploiting process similarity of 3D flash memory for high performance SSDs. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 211–223. ACM, 2019.

[57] Seungwoo Son and Jaeho Kim. Differentiated protection and hot/cold-aware data placement policies through k-means clustering analysis for 3D-NAND SSDs. *Electronics*, pages 398–412, 2022.

[58] The SSD Guy. Comparing wear figures on SSDs. https://thessdguy.com/comparing-wear-figures-on-ssds/, 2017.

[59] Theodore Ts'o. Ext2/3/4 file system utilities. https://e2fsprogs.sourceforge.net/, 2009.

[60] Vasily Tar Asov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. https://www.usenix.org/system/files/login/articles/login_spring16_02_tarasov.pdf, 2016.

[61] Fan Xu, Shujie Han, Patrick P. C. Lee, Yi Liu, Cheng He, and Jiongzhou Liu. General feature selection for failure prediction in large-scale SSD deployment. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 263–270. IEEE, 2021.

[62] 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. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 15–28. USENIX, 2017.

[63] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. AutoStream: automatic stream management for multi-streamed SSDs. In *Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR)*, pages 1–11. ACM, 2017.

[64] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 191–208. USENIX, 2020.

[65] Kong-Kiat Yong and Li-Pin Chang. Error diluting: Exploiting 3-D NAND flash process variation for efficient read on ldpc-based SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pages 3467–3478, 2020.

[66] Yuqi Zhang, Wenwen Hao, Ben Niu, Kangkang Liu, Shuyang Wang, Na Liu, Xing He, Yongwong Gwon, and Chankyu Koh. Multi-view feature-based SSD failure prediction: What, when, and why. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 409–424. USENIX, 2023.

[67] Hao Zhou, Zhiheng Niu, Gang Wang, Xiaoguang Liu, Dongshi Liu, Bingnan Kang, Hu Zheng, and Yong Zhang. A proactive failure tolerant mechanism for SSDs storage systems based on unsupervised learning. In *IEEE International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2021.

[68] You Zhou, Qiulin Wu, Fei Wu, Hong Jiang, Jian Zhou, and Changsheng Xie. Remap-SSD: Safely and efficiently exploiting SSD address remapping to eliminate duplicate writes. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 187–202. USENIX, 2021.

A Artifact Appendix

Abstract

As introduced in the paper, the current storage system abstraction of fixed capacity exacerbates aging-related performance degradation for modern SSDs, and enabling capacity variance allows for more effective tradeoffs between capacity, performance, and reliability. This artifact includes the code and describes the steps for measuring and comparing the performance of the proposed capacity-variant storage system against the traditional storage system to support our major claims. The experiments are performed on a machine with 32 CPUs and 1 TiB of memory running Ubuntu 20.04 LTS.

Scope

The provided code and scripts facilitate the testing of the following experiments:

- The performance degradation caused by aging observed on a real SSD (Figure 1).
- The functionality of CVSS, including CV-FS, CV-SSD, and CV-manager.
- The FIO experiments (Figure 10, Figure 11, and Figure 13).
- The Filebench experiments (Figure 12).
- The Twitter traces experiments (Figure 14).
- The lifetime experiments (Figure 15).

Contents

A.0.1 Fail-slow Experiments (Section 1)

Scripts are provided to age the SSD and measure its read-only I/O performance. To initiate the experiment:

```
$ ./fio_aging.sh
```

Note that the content of the tested SSD will be wiped out by the above script. The estimated time for this experiment depends on the endurance of the tested device and it may take several months to fully age the device.

A.0.2 Installation of CVSS

This section describes the steps to set up CVSS and perform basic tests. The `REMAP` interface is implemented on Linux kernel v5.15. To compile the kernel:

```
$ make -j$(nproc) bindeb-pkg
```

The CV-FS is configured as a kernel module. To compile and install the CV-FS:

```
$ ./run.sh
```

The CV-SSD is based on FEMU. To compile the code and start the virtual machine, please run the following commands after cloning the repository:

```
$ cd FAST24_CVSS_FEMU
$ mkdir build-femu
$ cd build-femu
$ cp ../femu-scripts/femu-copy-scripts.sh ./
$ ./femu-copy-scripts.sh ./
$ ./run-blackbox.sh
```

This will start the virtual machine with the emulated CV-SSD. You can set the path to your VM image via `IMGDIR=/path/to/image` in the `run-blackbox.sh` file.

A.0.3 Basic Test

To test the functionality of CVSS:

```
$ inscvfs
$ diskcvfs
$ df -h /dev/nvme0n1
```

The logical capacity of CVSS can be adjusted online by issuing the following command:

```
$ sudo cvfs.f2fs /dev/nvme0n1 -t 118
```

The parameter for the `-t` flag (e.g., 118) is the newly configured logical capacity in GiB that we have set for the system.

A.0.4 Evaluation Workflow

FIO experiments (Section 5.2.1). To evaluate the performance of CVSS under FIO-related workloads, please run:

```
$ ./test_fio_zipfian_util30.sh
$ ./test_fio_zipfian_util70.sh
$ ./test_fio_random_util30.sh
$ ./test_fio_random_util70.sh
```

Each experiment may take 4 days to finish. The virtual machine will be turned off when the experiment finishes, and the performance results will be stored in `.log` files in the working directory.

Filebench experiments (Section 5.2.2). To perform the filebench-related experiments, please run:

```
$ ./fs_test.sh
```

This script will age the system and issue *Fileserver*, *Netsfs*, and *Varmail* workloads under different aged states of the underlying device. The latency results are logged in `.log` files in the working directory.

Twitter traces experiments (Section 5.2.3). To set up RocksDB and issue Twitter traces to the system, please run:

```
$ cd ./rocksdb/examples
$ gcc twitter_load.c -o twitter_load
$ gcc twitter_run.c -o twitter_run
$ ./twitter.sh
```

Each test may take one week to complete. The IOPS and trace profiles are stored in the `.log` files.

Lifetime experiments (Section 5.3). To test the amount of host writes under different performance requirements and workloads, please run:

```
$ ./test_lifetime_zipfian_util30.sh
$ ./test_lifetime_zipfian_util70.sh
$ ./test_lifetime_random_util30.sh
$ ./test_lifetime_random_util70.sh
```

Each experiment is expected to take approximately 5 days to complete. The experiments will continue running until the underlying SSD fails. Performance results are documented in the `.log` files within the working directory. Additionally, device statistics, such as the write amplification factor, can be found in the `wa.log` file located in the host directory of the virtual machine.

Hosting

The artifact is available on github repositories:

- Kernel: https://github.com/ZiyangJiao/FAST24_CVSS_Kernel.
- CV-FS: https://github.com/ZiyangJiao/FAST24_CVSS_CVFS.
- CV-SSD: https://github.com/ZiyangJiao/FAST24_CVSS_FEMU.

Requirements

Please make sure you have at least 160 GiB of memory and 150 GiB of free space on your disk if testing on your machine. Our evaluation is based on the following system specifications:

Component	Specification
Processor	Intel(R) Xeon(R) Silver 4208 CPU, 32-Core
Architecture	x86_64
Memory	DDR4 2666 MHz, 1 TiB (64 GiB x16)
SSD	Intel DC P4510 1.6TiB
OS	Ubuntu 20.04 LTS (Focal Fossa)

I/O in a Flash: Evolution of ONTAP to Low-Latency SSDs

Matthew Curtis-Maury, Ram Kesavan*, Bharadwaj V R*, Nikhil Mattankot, Vania Fang,
Yash Trivedi, Kesari Mishra†, and Qin Li
NetApp, Inc

Abstract

Flash-based persistent storage media are capable of sub-millisecond latency I/O. However, a storage architecture optimized for spinning drives may contain software delays that make it impractical for use with such media. The NetApp® ONTAP® storage system was designed originally for spinning drives, and needed alterations before it was productized as an all-SSD system. In this paper, we focus on the changes made to the read I/O path over the last several years, which have been crucial to this transformation, and present them in chronological fashion together with the associated performance analyses.

1 Introduction

The advent of flash-based storage about a decade ago transformed the business of data center storage controllers. Despite improvements in several dimensions, the time to access any randomly selected data from storage had historically remained limited by physical constraints of spinning hard disk drive (HDD) technology. NAND-based solid state drives (SSDs) provided orders of magnitude lower latency and higher IOPS. In the last decade, several SSD-optimized or SSD-only architectures for data center storage controllers have been built and productized.

NetApp's® flagship feature-rich ONTAP® storage operating system is deployed in various configurations both within the data-center and in the cloud. ONTAP and its proprietary WAFL® file system [19] were optimized over their first two decades to maximize available I/O bandwidth on HDDs (with multi-millisecond latencies) for both reads and writes, and with a modular architecture to allow ongoing feature development. Most features of the WAFL architecture are required of any enterprise-quality storage system regardless of the underlying persistent media: data integrity [38], availability, data protection [51], recovery [28], etc. WAFL metadata was designed to optimize random metadata lookups from media, efficiently write out data and metadata to storage [25, 29], and to enable key functionality such as snapshots. Compression and deduplication techniques in WAFL improved efficiency in storage capacity. The WAFL consistency point converted random updates of user data and metadata into sequential I/O [13, 30], and wrote blocks to areas with the most free space, which turned out to be well-suited to minimizing FTL write amplification in SSDs.

ONTAP also integrated flash technology—PCIe-attached Flash Cache® [54] and SSD tiering in Flash Pool® [55]— but due to software delays in the ONTAP legacy data path, applications benefitted only partially from SSD's sub-millisecond latency, particularly for random reads. As such, ONTAP was faced with the challenge of making I/O software overhead commensurate with device latency in order to ship a competitive all-SSD system. Other legacy storage systems have similarly found software overhead out of proportion to low-latency device access times [8, 24, 32, 33, 40, 57, 62, 71]. Building a new storage architecture "from scratch" for SSDs would have required reinventing dozens of battle-tested features that were critical to our enterprise customers. As noted above, ONTAP and WAFL already had most of the building blocks necessary for building an all-SSD controller. Therefore, we instead reworked the legacy read path to speed it up incrementally over several software releases spanning multiple years, primarily by eliminating message hops between components of the storage software stack and moving towards a run-to-completion execution model that minimizes expensive message passing steps.

Although this paper focuses only on the optimization of the read I/O path, a collection of other improvements were also crucial to productizing the all-SSD controller. As described in prior work [25], we changed the block allocator to write contiguously down the SSD LBA-space in multiples of the SSD erase page size, thereby mitigating the log-on-log problem [67] and increasing SSD lifetimes. We redesigned the compression and deduplication infrastructure to run *inline* with writes, which reduces the overall data written to storage thereby further prolonging SSD lifetimes. We introduced other key performance optimizations, including in the write I/O path and journal replay.

This paper makes the following contributions: We analyze the latency breakdown of the legacy read path of a success-

*Currently employed at Google, †Currently employed at Meta

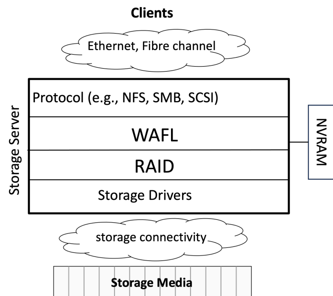


Figure 1: ONTAP modules involved in the data path.

ful enterprise storage system. We present a series of performance improvements made by systematically removing the primary sources of software delay. We analyze the improvements using data from detailed experiments across a range of hardware platforms and a cloud-resident VM. Finally, we discuss two major lessons that we learned from our experiences. Our improvements dropped software overhead from multiple milliseconds to less than 160us (more than 20X) generating large improvements in read latency and throughput, which has been foundational to the all-SSD ONTAP controller becoming a multi-billion dollar product line.

2 Background

In this section, we provide a brief overview of ONTAP and WAFL followed by a description and analysis of the legacy read path. We refer readers looking for a deeper understanding of WAFL to prior work [12, 13, 15, 19, 25, 26, 28, 29].

2.1 ONTAP Storage Stack

Fig. 1 shows the major ONTAP components in the data path. The Protocol component receives requests from clients and converts them into WAFL requests. The WAFL component processes all requests to the file systems—I/Os, operations related to data management, replication, etc. The layers beneath provide access to the storage media and implement RAID protection across them. Each component has data structures that are accessible typically only from thread pools dedicated to the component, which simplifies the synchronization between components. Component boundaries are traversed by message-passing between their threads, which means a request from a client undergoes multiple hops.

2.2 WAFL Processing Model

ONTAP houses and exports multiple file systems called volumes from within a shared pool of storage called an *aggregate*, and the WAFL component handles operations on them. The WAFL file system stores all metadata and user data in files which are organized in a hierarchical fashion. WAFL blocks are 4KiB in size and alignment, and are indexed in the aggregate by a PVBN (physical volume block number). Detailed descriptions are available elsewhere [15, 19].

Requests are dispatched to the WAFL component as WAFL messages. All data in the file systems are conceptually arranged into a hierarchy of data partitions called *affinities*, in a model referred to as *Waffinity* [12]. Based on its type and the data it intends to access, each request is dispatched to a specific affinity in the hierarchy. A dedicated pool of Waffinity threads execute requests on a per-affinity basis within the WAFL component in a thread-safe fashion. Each message type has an associated handler, which is coded in a *load-modify* transactional model: all resources necessary for the operation are accumulated in the *load* phase during which the message may suspend one or more times, after which the handler is completed in a single non-blocking *modify* phase, during which any mutations to the file system state are committed. This execution model together with the guarantees of Waffinity ensures that WAFL operations execute in atomic fashion with parallelism-safety.

If a resource is unavailable, the message releases all resources acquired thus far before it suspends (blocks) on an appropriate wait-list, thereby avoiding resource dependencies and deadlocks. When woken up, the message handler restarts execution from the beginning to try and reacquire the necessary resources. For example, a read message requires that the data blocks are available in memory. If those blocks are not in memory during the load phase, the read handler initiates retrieval of those blocks from persistent storage and suspends awaiting that retrieval. Upon restart, it goes through the same steps, but likely finds the blocks in memory this time and is able to complete its modify phase. This model provides deadlock-free concurrent execution but trades off CPU cycles for potential load phase re-execution.

2.3 Mutations to the File Systems

ONTAP was always designed to process mutations to the file system state with low latency. Consider the example of a write request. The load phase of the WAFL write message handler ensures the necessary inode and ancestor indirect blocks are in memory. The modify phase updates the file system state in memory and journals the write to NVRAM[1] before it responds to the Protocol component, which then sends an acknowledgement to the client. The modify phase leaves behind "dirty" file system state in memory—inodes, buffers, and volumes. Dirty state is collectively and periodically persisted on a per-aggregate basis as a single transaction called a *consistency point* (or *CP*) [13, 25]. Dirty data is compressed, deduplicated, and compacted [27] asynchronously to the write but before the CP completes. Because the inode

[1]ONTAP systems are deployed as HA-pairs, and the journaled write must get mirrored to the HA partner's NVRAM as well.

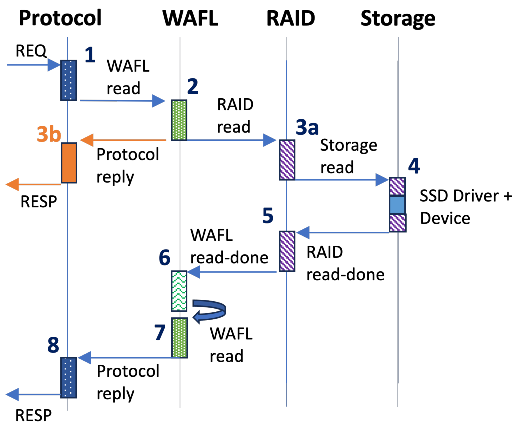


Figure 2: Message passing steps across ONTAP components for a read request.

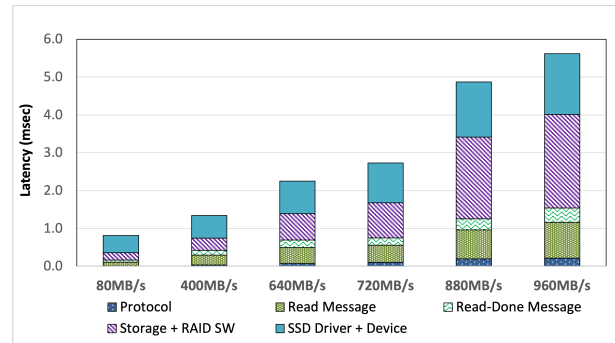


Figure 3: Latency across ONTAP components with increasing load for a 4KiB random read request.

and ancestor blocks of "hot" byte ranges are typically memory resident and appends to NVRAM are fast, the write latency is mostly independent of storage access times. Therefore, the read path was the main focus of the performance work required for productization of the all-SSD controller.

2.4 Legacy Read Path

Fig. 2 shows the message passing hops in the legacy read path and is applicable to all supported protocols—SCSI, NVMe, NFS, and SMB. In step 1, ONTAP receives the read request over the network in the context of a Protocol thread, which parses the request and translates it to a WAFL read message. A Waffinity thread picks up and executes the message in step 2. The read handler traverses file system data structures to find the requested data blocks. If the required data are found in memory during the load phase, the data is assembled into a reply payload in the modify phase and sent back to the Protocol thread, which replies to the client in step 3b. If not, the handler suspends until the data is available in memory. One such example is when all required intermediate data—inode, indirect blocks, etc.—are found, but the data blocks are not. In this case, the handler allocates, initializes, and inserts one buffer per missing block in the file system tree, places them in one or more RAID read messages that it sends to the RAID component, and suspends the WAFL read message on the completion of all required I/Os. WAFL uses the PVBNs of the blocks to determine the required number of RAID read messages.

A RAID thread processes each RAID read message, uses its knowledge of the drive mappings to translate each PVBN to drive ID and LBA, and sends a message to the Storage component in step 3a. In step 4, a Storage thread processes this message, dispatches a read to the physical drive, and sends a read-done to the RAID component upon completion of the I/O. In step 5, RAID validates checksums and sends a read-done to the WAFL component if no errors are found. In step 6, a WAFL thread does further validation[2], marks the buffers *valid*, and restarts all waiters. The original WAFL read message is awoken once all issued RAID messages (from step 2) have completed. In step 7, a Waffinity thread runs the original message by re-executing the handler, eventually finds all valid buffers in cache, and replies to the Protocol component, which replies to the client in step 8.

Three different data reduction techniques—compression, deduplication, and sub-block compaction [27]—are used in combination by WAFL to efficiently store data; the data is also encrypted just before it is stored. Decryption occurs in step 4 while reading from storage, but the choice of where (in one of the steps in the reply path) the reduced data gets rehydrated is made dynamically based on various conditions. We consider the topic of data reduction outside the scope of this paper for two reasons: it is too large a topic to cover comprehensively, and it would be a distraction because the techniques and results presented in this paper are fundamentally unchanged with or without data reduction.

This architecture is modular, which facilitates continuous feature development, and error handling can be performed in the corresponding component. A WAFL read that does not hit in the buffer cache may incur several message hops including multiple suspensions and restarts within WAFL before completion. Such hops become expensive under CPU pressure, when a message must wait for the next thread to be scheduled or when running threads cannot keep up with incoming load. In the case of HDDs, these scheduling delays are typically dwarfed by drive I/O latencies. Such delays become noticeably large for SSDs.

2.5 Components of Read Latency

Fig. 3 shows the breakdown of *server-side* latency for a read request to ONTAP across the steps outlined in Fig. 2 under

[2] WAFL stores a *context* together with each written block [60] to identify its file and offset to protect against lost or misdirected writes [3], and identify a block that has been moved for defragmentation purposes [26].

increasing levels of load, using a matching color scheme for each step. This data was collected on ONTAP 8.2.2 (circa 2014), which predates the optimizations discussed in this paper. A random read workload—which ensures a low buffer cache hit rate and frequent drive access—was run on a 2x10-core Intel Xeon 2.8GHz controller with 128 GiB of DRAM, the high-end ONTAP system from that time. The controller had twelve 400GiB SAS SSD drives, which collectively provided sufficient I/O throughput for the highest load of this experiment. A set of LUNs were configured on ONTAP and a number of clients sent 4KiB reads to random offsets using the FCP storage protocol over an underlying Fibre Channel network to cumulatively create the desired load. Throughout this paper, time within each component is measured using start/stop timers in the software stack. Network component time is included within the Protocol layer, which collectively remain a small source of delay due to their relative efficiency compared to other components in the stack.

Protocol corresponds to steps 1 and 8. *Read Message* and *Read-Done Message* depict time in the WAFL component, the former for the sum of steps 2 and 7 and the latter for step 6. *Storage+RAID SW* corresponds to steps 3, 4, and 5 minus *SSD Driver+Device*, which depicts the latency in the device driver and media. The raw CPU cycles in the WAFL and Protocol components (steps 1, 2, 6, 7, and 8) are negligible (each less than 30us); in other words, most of the latency is the message waiting to be picked up by WAFL or Protocol threads. Although *SSD Driver+Device* time does increase with load, all of that increase is attributed to software delays in the device drivers due to increased CPU wait times. We confirm this later in Fig. 7, which shows consistent *SSD Driver+Device* times when CPU wait times are not a major factor. Increased load amplifies the cost of each hop because threads are busier and CPUs are closer to saturation.

At 80% of maximum throughput of the system (960MiB/s), the device latency is less than 30% of the total read latency. The primary non-device delays are in the RAID/Storage components and wait times in WAFL for the read and read-done messages. While such delays were acceptable for HDDs with media latency of several milliseconds, their impact became outsized for SSDs with media latency of a few 100's of microseconds. Clean-sheet design approaches for the read path were discarded because of the inherent complexities around handling myriad error conditions and integrating with existing ONTAP features. Instead, we used the latency data to iteratively improve the read path for the most common cases while retaining the legacy path for error handling and other complicated conditions.

3 Fastpaths: WAFL Reply and RAID

Optimization of the read path for SSDs started as a skunkworks project in the WAFL team, and we began with the WAFL reply path. Because the latency breakdown was dominated by wait-times due to message passing hops, we chose to eliminate hops, steps 6 and 7, instead of optimizing code. We call this work *WAFL Reply Fastpath*. Next, the RAID and Storage teams eliminated steps 3a and 5, called *RAID Fastpath*. These changes are shown in Fig. 4.

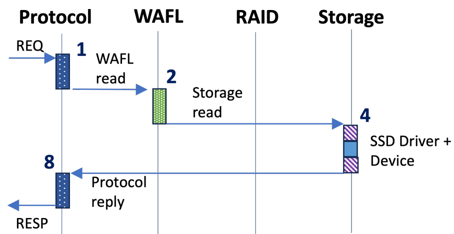


Figure 4: Steps with WAFL Reply and Storage Fastpaths.

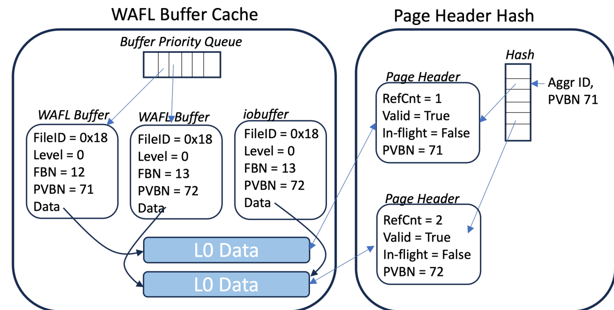


Figure 5: WAFL buffer cache and page header hash.

3.1 Bypassing WAFL Read-done

The WAFL read-done message validates the data, updates the WAFL buffer state to reflect the I/O completion, and restarts the original WAFL read message. We explored whether the error-free path of this handler could be executed directly by the RAID component as part of RAID read-done (step 5). We refer to this technique as *bypassing layers*. Data blocks and indirect blocks of a file are represented in memory as *WAFL buffers*, which are logical headers that point to 4KiB *data block pages*. WAFL buffers are arranged into per-file inode block trees. A multi-level LRU structure [14], labeled Priority Queue in Fig. 5, tracks the aging and priority of buffers, and is designed to be accessible from within and outside of the WAFL component as a result of earlier performance improvement work.

A given data block can be shared by several inodes' block trees; this capability is used by many ONTAP features, such as snapshots, deduplication, cloning, etc. Hence, multiple WAFL buffers can point to a data block page. Each block page also has a statically-associated *page header*, that stores metadata about the page such as a reference count of the WAFL buffers pointing to it. The page headers are tracked in a header hash indexed by aggregate ID and PVBN, which

is looked up before issuing I/O. Fig. 5 shows two example block pages with PVBNs 71 and 72. Access to block pages and page headers are protected by range locks on the page header hash from any component. A block page can only be scavenged when its page header refcount is zero, which implies all buffers pointing to it have been evicted.

In the legacy read path, a WAFL buffer (per block) was sent with the RAID read message. The RAID and Storage components could safely update certain flags/fields in those buffers to track I/O state, error states, the checksum, etc. However, marking the buffer valid could happen only within the WAFL component, hence the need for WAFL read-done. In the new model, we add a valid state in the page header and leverage a new *iobuffer* object that is used exclusively for the purpose of I/O and is therefore exempt from many of the rules that govern WAFL buffers. As in the legacy path, the WAFL read message inserts a WAFL buffer but now also initializes an iobuffer per block, which it instead sends with the RAID read message. The iobuffer is private to the read request and cannot be found otherwise. Both buffers point to the same block page, as shown in Fig. 5. The WAFL read message now suspends on a page header (instead of a WAFL buffer) waiting for it to become valid. The RAID read-done message first validates the data block then directly invokes a WAFL function that performs the file system specific validation, marks the page header valid, wakes up the suspended WAFL read message, and frees the iobuffer. If it encounters any errors, it can safely fail through to WAFL at any point, because WAFL messages always restart execution from the beginning of the message handler.

3.2 Bypassing the Restart of WAFL Read

The removal of step 6 resulted in significant improvements, and encouraged us to next explore eliminating step 7. In the legacy read path, the restarted WAFL read message ensured that all data was present in memory, assembled them into a vector, and replied to the Protocol component. As with the WAFL read-done message, this work is now executed inline by the RAID read-done (step 5) message by using iobuffers. The original WAFL read message is attached to the RAID read message, in which we keep count of the outstanding I/Os to storage. This count is atomically decremented upon each I/O completion, and the last completion replies to the Protocol component. If any errors are encountered, the legacy path is triggered by sending the read message back into WAFL. When a Protocol thread receives the reply, the embedded WAFL read message is freed. The original WAFL buffer is marked valid only if accessed by some subsequent WAFL message (or the restarted WAFL read message in case of an error) on finding that the buffer points to a valid block page. If never accessed, the buffer eventually ages out like any other. The elimination of steps 6 and 7 is collectively called the WAFL Reply Fastpath.

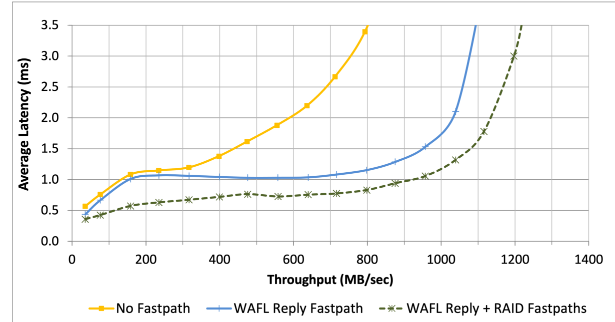


Figure 6: Latency vs achieved throughput with increasing 4KiB random read load with and without Fastpaths.

3.3 RAID Fastpath

We next worked to bypass the RAID component (steps 3a and 5) entirely on the read path. The RAID component maintains an up-to-date topology data structure of the aggregate; it knows which drives are in some failure state or are getting reconstructed. RAID uses that information in RAID read (step 3a) to map the PVBN of each buffer supplied by WAFL to the physical drive and LBA. RAID exports a read-only cache of the topology, which is now used by the WAFL read message for the translations and to directly send I/O messages to the Storage component. Changes in the aggregate, such as addition of drives, failure of drives, or RAID reconstruction, will require updating the topology. Though rare, when such events occur the RAID component flags the cache as stale, and the WAFL read message fails through to using the legacy RAID read. In the case of a race—say the topology is tagged stale after a Fastpath is triggered—the Storage component detects the staleness in step 4 and returns an error, and the restarted WAFL read message now fails through to the legacy path. The Fastpath resumes once a new topology cache has been built and exported by RAID.

Upon completion of a device I/O in step 4, the Storage component now directly calls a thread-safe version of the checksum validation used in RAID read-done (step 5), followed by the WAFL Reply Fastpath described in Sec. 3.1 and Sec. 3.2. As elsewhere, the legacy path is used as a fail-safe whenever any error is encountered.

3.4 Performance Analysis of Fastpaths

Fig. 6 and 7 show results from the same 4KiB random read experiment on the same 20-core platform from Sec. 2.5 with the Fastpaths enabled. *No Fastpath* data was collected by using ONTAP 8.2.2 (circa 2014), which precedes our optimizations, *WAFL Reply Fastpath* using ONTAP 8.3.0 (early 2015), and then with *RAID Fastpath* using ONTAP 8.3.1 (late 2015). Although not strictly apple-to-apples because we are comparing different releases, the performance impact seen in these graphs is primarily due to the Fastpaths. Fig. 6

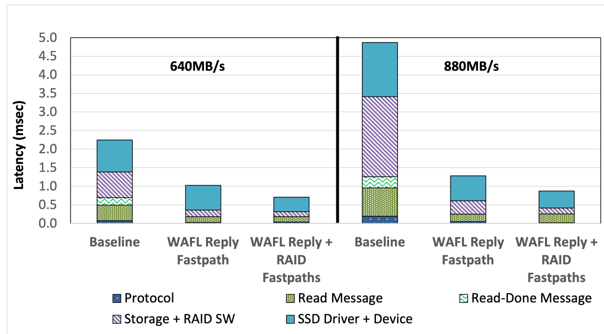


Figure 7: Latency across ONTAP components with and without improvements at two specific loads of 4KiB random reads.

plots the average server-side latency vs achieved load, and shows how these improvements have significantly shifted the system saturation points to the right. The read throughput at the average latency of around 1ms (the industry expectation for SSD-controller latency in the mid-2010's) quadrupled from 150MiB/s to 600MiB/s with WAFL Reply Fastpath, and increased another 50% from 600MiB/s to 900MiB/s with RAID Fastpath. Fig. 7 shows the latency breakdown at two specific load points. The sharp increase in latency with the legacy path is attributable primarily to the RAID and Storage components. From mining finer grained statistics in ONTAP, the savings at 880 MiB/s compute to 300us of wait time for the WAFL read-done message, 480us wait time for the restarted WAFL read message, and a smaller 17us of CPU time across both messages. More interestingly, the reduction in CPU utilization due to the elimination of steps 6 and 7 results in lowered wait times for threads in all components, lowering the overhead of remaining message hops and deferring CPU saturation to higher levels of load. Adding the RAID Fastpath at 880 MiB/s results in a further reduction in wait times in RAID and Storage components and a reduction in device driver wait time. In the end, software overhead is on par with device times.

It should be noted that latency variance in ONTAP is almost always due to variance in wait times, which gets worse only with increased CPU utilization. Therefore, latency variance is high only to the right of the "knee" of the latency-throughput curve [50]. Because Fastpaths (and the improvements presented subsequently in this paper) significantly reduce wait times, their benefits for p90 and p99 latencies have an outsized impact to the right of the knee of the curve. For example, p90 latency drops from 7ms (for legacy) to 2ms (with both Fastpaths) in this experiment. Therefore, we use average latency as a conservative showcase of the improvements throughout this paper.

In this section, we presented and evaluated a collection of optimizations to minimize message hops in the read I/O path. We showed that component layers can be effectively bypassed by running limited elements of one layer within

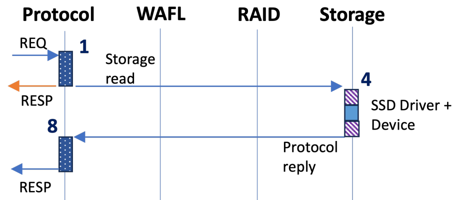


Figure 8: Read path steps with TopSpin read design.

another layer to constrain software overhead. This work was crucial to NetApp shipping a feature-rich all-SSD ONTAP controller in 2015 instead of creating an SSD-optimized file system from scratch. Further, the success from Fastpaths encouraged the continued use of this approach in ONTAP. The next section discusses how we used bypassing to tackle the dominant remaining delay.

4 TopSpin Read: Bypassing WAFL Read

By the early 2010s, it was obvious that traditional interconnects such as SAS and SATA were inadequate for SSD speed and bandwidth. Based on the new NVM Express technology [52], enterprise-quality SSDs boasting at least one order of magnitude better performance were available by the late 2010s. NVMe storage drivers were added to ONTAP to access these new SSDs. Additionally, Linux and Windows clients were now able to unlock these performance benefits by connecting over the network using the NVMe over Fabrics (NVMe-oF) protocol. In response, an NVMe server-side module optimized for parallelism and low latency was added to the ONTAP Protocol component. With these technological improvements, tackling the remaining large software delay—the wait time for WAFL Read (step 2) as seen in Fig. 7—became a competitive imperative.

To that end, we developed *TopSpin*, an optimization to allow the common-case read request to bypass the WAFL component, as shown in Fig. 8. TopSpin leverages direct access to WAFL data structures from within the Protocol component (step 1) to check if the required data is in memory and to issue I/Os to storage, while handling all potential races with requests that modify file system state running in parallel within the WAFL component. This design has three advantages: (1) It avoids the queueing delays within WAFL. (2) The reimplemented read handler is light-weight and avoids the suspend-restart CPU overhead. (3) It bypasses the strict data partitioning within WAFL that can restrict parallelism. It was productized in ONTAP 9.3 (2017) and enabled for all block-based protocols—SCSI and NVMe.

4.1 Storage Location Cache

We introduce the Storage Location Cache (SLC) to allow the Protocol component to directly and safely discover data lo-

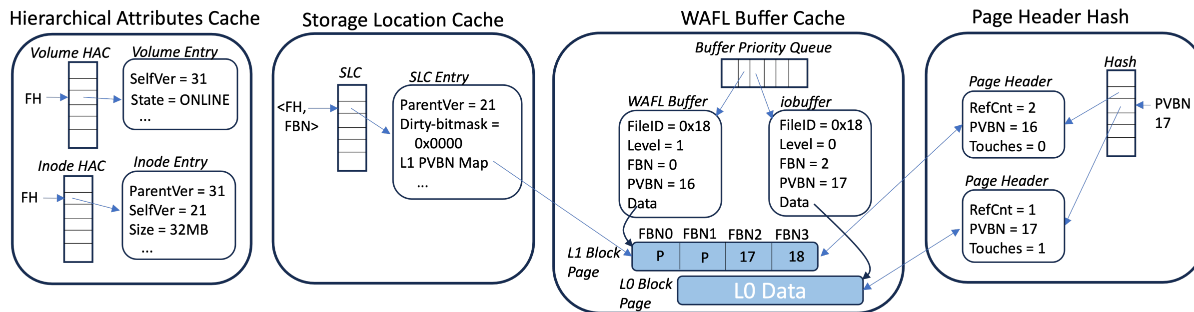


Figure 9: Overall SLC and HAC architecture, which integrates with the existing WAFL buffer cache and page header hash.

cations. The SLC is a hash table that maps file handle and file block number (or FBN, the 4KiB file offset) to PVBN; its hash buckets are protected by range locks. The lowest level indirect blocks in a WAFL inode tree (Level-1 blocks, or *L1s*) comprise this map, along with per-FBN auxiliary information used for data validation. The location of the ith FBN is found in the $(i\%span)^{th}$ index of the $(\frac{i}{span})^{th}$ L1, where the fixed span is the maximum number of children an L1 can have. As Fig. 9 shows, each SLC entry points directly to a block page of one L1 and the SLC entry takes a refcount through the corresponding page header. SLC entries are inserted (when an L1 block page is loaded into memory) and updated only from the WAFL component, including being removed when the L1 page is scavenged.

4.2 Hierarchical Attributes Cache

For a read request to be safely processed in the Protocol component, it must synchronize with changes to file system state occurring in parallel within the WAFL component. For instance, changes to the mount state of a volume or the size of a file may interact with a read request. We introduce the Hierarchical Attributes Cache (HAC) to track properties of file system objects—inodes and volumes—to facilitate such checks. Each user file or LUN is represented by an HAC inode object that caches various attributes of the inode, such as size and permissions. Each volume is represented by an HAC volume object that caches mount state, encryption key, etc. As Fig. 9 shows, the objects are organized into two hash tables indexable by file handle (which includes a volume identifier). Access to these objects is protected by a lock per hash bucket. Much like the SLC, HAC objects are consulted from the Protocol component but created and updated only from the WAFL component; a volume (inode) object is added to the HAC when it is mounted (loaded into memory).

4.3 TopSpin Read from Protocol Component

We implement *TopSpin read*, a version of the WAFL read handler that is called directly by the Protocol thread towards the end of step 1. Fig. 9 shows the system state for an example TopSpin read of FBN2 and FBN3 of a file. It first looks up the SLC using file handle and offset to determine if the requisite L1 block pages are in memory; the actual lookup converts the offset to the FBN aligned to L1 span, which is FBN0 in this case. If found, it confirms the freshness of the SLC entries by consulting the HAC inode and volume objects; Sec. 4.5 details the the freshness check. Next, it indexes the L1 page to obtain the PVBNs (and auxiliary information), 17 and 18 in the figure, and looks them up along with aggregate ID in the page header hash. If all block pages are found in memory, it inserts them into the reply vector and replies to the client, holding a page refcount until complete. Otherwise, much like the WAFL read handler, TopSpin uses the PVBN and auxiliary information to instantiate iobuffers, block pages, and page headers, and sends the appropriate I/Os to the Storage component for the missing Level-0 file data blocks (or *L0s*). The Protocol component resumes processing this request after receiving a reply from Storage, much as in Sec. 3.3. If TopSpin read fails for any reason, such as missing L1 block pages or freshness check failure, it falls through to the legacy WAFL path. Both caches—HAC and SLC—use LRUs to age their entries, thereby increasing the chances of TopSpin reads to "hot" file byte ranges completing successfully.

As noted in the Fastpath sections, some data structures were already safe to access from outside the WAFL component—the block pages, page headers and hash, the RAID topology cache, etc. TopSpin limits itself to accessing only those shared structures. When an I/O completes, step 4 now inserts the iobuffer directly into the buffer cache LRU, unlike in the Fastpath case where the iobuffer is discarded and the WAFL buffer is preserved. TopSpin reads access L0 block pages directly through the page header hash, and increment a newly added *touches* count field in the page header to track hotness (shown in Fig. 9). Before an iobuffer can be scavenged, any such references are transferred from the corresponding page header into the iobuffer thereby preventing eviction. We next look at how the SLC and HAC guarantee correctness.

4.4 Keeping SLC Consistent

A write request executing in parallel within the WAFL component may conflict with a TopSpin read, and we use the SLC entry for synchronization. Each SLC entry uses a *dirty-bitmask* to track whether its children data blocks have been "dirtied" by any operation running in the WAFL component, one bit per child. In its modify phase, the WAFL write handler locks up to 3 SLC entries—the largest write supported (1 MiB) may span up to 3 L1s—to set the dirty bit for each FBN. A TopSpin read looks up the dirty bits after locking the necessary SLC entries, and fails through to WAFL if any is set. Otherwise, it obtains the PVBNs and either finds the block pages through the page header hash or issues I/Os.

The subsequent CP walks through each dirty buffer and allocates a new location for it in storage, a previously free PVBN. Then, it rehashes the dirty block page using the new PVBN in the page header hash, after which the page is sent together with several other pages as a write I/O to a RAID group in the aggregate; more details in other work [13, 25]. In theory, each dirty bit in an SLC entry can be cleared when the CP rehashes the child L0 block page with the new PVBN—a subsequent TopSpin read can now safely read that block page. Instead, to amortize locking costs, all dirty bits in an SLC entry are cleared together by locking the SLC entry just once after the CP is done with the L1 and all its children. It typically takes anywhere from 2-5 seconds for a subsequent CP to process that parent L1 and clear the dirty bits in the SLC entry. This is rarely a problem for our customer environments, where immediate reads after writes are rare, but would result in failing through to WAFL.

4.5 Keeping HAC Consistent

SLC entries may be invalidated by various infrequently occurring operations, such as a volume remount or a file resize. These are tracked by versioning HAC objects. Each HAC object records two version numbers: a self version v_s for its child relationships and a parent version v_p for its parent relationship; Fig. 9 refers to them as SelfVer and ParentVer, respectively. The former is initialized when an object is created and incremented when any of its attributes change. For example, if a file is resized its inode object's v_s gets incremented. Each SLC entry also records a v_p. A hierarchy exists: each SLC entry has a parent inode, and each inode HAC object has a parent volume HAC object. When an HAC object is created (updated), its v_p is initialized to its parent's v_s and its own v_s is set (incremented). An object or entry is confirmed to be fresh only if its v_p matches its parent object's v_s. A check must recurse up the hierarchy to confirm freshness.

A failed SLC entry check at the inode (or volume level) indicates that the corresponding file (or volume) has since been modified in some way that makes the SLC entry stale. When that happens, TopSpin sends the read to the WAFL component. Stale SLC entries and HAC objects age out of their respective caches. Version numbers are incremented only by operations running within the WAFL component. Incrementing the version of an object implicitly invalidates all its descendent objects, which may be numerous—a volume may comprise hundreds of files, each with thousands of "hot" L1s. It should be noted that version bumps occur infrequently, and therefore the fast 3-level recursive check done by a TopSpin read succeeds most of the time.

4.6 TopSpin and File-based Protocols

The improvements in Sec. 3 moved portions of the read path from WAFL and RAID down to the Storage component, and are therefore independent of the client protocol. All protocols can benefit from the Fastpaths. In contrast, TopSpin read requires changes to the code in the Protocol component. We implemented TopSpin first for block-based (SAN) protocols because SAN applications—such as databases, server virtualization, and business applications—require consistently low latency. NAS protocols require that a read check other metadata, such as file permissions, ACLs, and lock state. These structures are currently accessible only from within the WAFL component. An inode's access time (*atime*) also needs to be updated on reads, which results in mutations that need to be persisted. A TopSpin read would need to safely access the corresponding structures.

In this section, we extended Fastpath to bypass the WAFL layer in the read I/O path, by developing an alternative method for scalable, thread-safe file system accesses. Thus far, it has been narrowly deployed within ONTAP, but it can be applied to other protocols and file system operations. Extending TopSpin to NFS and SMB is a work in progress.

5 Client-Visible Consistency Semantics

In the previous sections, we discussed correctness in the face of race conditions within and between components. We now look at correctness from the client's point of view. Clients communicate with ONTAP using one of several protocols—NFS, SMB, SCSI, and NVMe-oF—each with its corresponding correctness semantics. To maximize code reuse and to simplify design and testing, ONTAP implements a conservative and consistent interpretation of the semantics across all protocols. Changing these interpretations is disallowed across ONTAP releases because it risks destabilizing client libraries and customer applications. All improvements presented in this paper preserve ONTAP's interpretations of these semantics. We look at only two rules that are relevant to this paper. In this section, we use the term "write" generically for any operation that mutates file system state and "read' for any that does not.

For example, if a client issues a read R after it has received the acknowledgement to a write W, then R must never

see any file system state prior to W. Because a server cannot know the exact moment when an acknowledgement is received by a client, we implement rules based on when requests (acknowledgements) enter (exit) the networking stack of the Protocol component. (1) *Read-After-Write* (RAW): ONTAP guarantees that once the Protocol component has issued an acknowledgement of a write, a read request received subsequently by the Protocol component sees only the state after the write. (2) *Concurrent-Read-Write* (CRW): If a read and write overlap in time when processed by ONTAP, the read sees state only from before or after the write, but never both; except for SCSI, where CRW applies only for sizes up to 64 KiB[3]. Both rules hold even if the read and write are sent by different clients using different protocols. In legacy ONTAP, every file system request was processed by the WAFL component in both the request and reply paths. The load-modify transactional model together with Waffinity guaranteed serialization of a read and a write if they conflicted in file byte range; this trivially satisfied both rules. The improvements presented in this paper are relevant only to read requests—one of the many possible "reads" as used generically in this section.

The Fastpaths avoid the RAID and WAFL components on only the reply path, so trivially preserve RAW. Because the results of a write are committed to the in-memory file system state by the WAFL component before its acknowledgement can be sent, it is impossible for a TopSpin read to see content from prior to the write. Thus, TopSpin also preserves RAW.

With both Fastpaths and TopSpin, if the write runs first then the read sees data from only after the write. In the case of TopSpin, the write first locks all SLC entries and sets the dirty bits in them, so the read fails the freshness check and falls through to WAFL. If the read runs first and finds all data in memory, it replies with data from only before the write in both Fastpaths and TopSpin. The TopSpin read locks all the SLC entries it needs. If the read runs first and finds that all its L0 pages are missing, it uses PVBNs from the L1s to issue I/Os to the Storage component. TopSpin finds the PVBNs via the SLC entries and the WAFL read handler via the L1 buffers. Even if a subsequent write dirties one or more of those FBNs, the read replies with only the persisted data prior to the write. Any read with a mix of hits and misses in the page header hash fails through to the legacy path. Although this case can be improved, it does not occur often in our customers' applications. Thus, CRW is preserved.

6 Performance Evaluation of TopSpin

A typical ONTAP controller hosts datasets for multiple instances of different applications that are accessed at the same time. No individual workload represents all outcomes in such multi-tenant environments. Therefore, we primarily used micro-benchmarks to study the performance, knowing that the results extend to any workload comprising those traffic patterns. We also tested with an in-house benchmark identical to the industry-standard SPC-1 [11], which models the query and update operations of an OLTP/DB application and simulates real world environments [17]. Lastly, we used a standardized load to Oracle. All experiments used a recent internal build based of ONTAP 9.13.1, unless otherwise indicated. We picked a *mid-range controller* with 2.2GHz Intel Xeon Silver 2x10 cores, 144GiB of DRAM, 16GiB of NVRAM, and 23 3.84TiB NVMe SSD drives to study the benefits when CPU resources are tight and a *high-end controller* with 2.2GHz Intel Xeon Platinum 2x32 cores, 1TiB of DRAM, 64GiB of NVRAM, and 46 3.84TIB NVMe SSD drives. The NVMe SSD drives support 100K IOPS of random reads with latency under 100us, and are configured into RAID double parity [10]. These configurations are realistic and are sufficient to make most workloads CPU-limited. A given IOPS load is collectively initiated in an open loop by a set of remote clients, such that queuing in the server becomes significant under heavy load. Latency is measured on the ONTAP server from when a read request enters to when its corresponding reply exits the controller. The TopSpin SLC is backed by L1 block pages in the WAFL buffer cache which can consume the majority of a system's DRAM and prioritizes indirect blocks. An L1 page in WAFL can reference 255 child blocks, so TopSpin can be effective with sizes significantly smaller than an application's working set.

Controllers are deployed as a high-availability pair, but we report only per-controller results. Only half of NVRAM is used by a controller because the other half is used to mirror the HA-partner's journal. Although compression, deduplication, and compaction [27] are now enabled by default on ONTAP all-SSD controllers, we disabled them in these experiments for three reasons: (1) Enabling them on datasets with realistic compressibility and dedupe savings does not change the character of the results. (2) We have not presented the designs of these data reduction techniques and how they interact with the read path. (3) We lack the space to explore the range of datasets that yield different combinations of compressibility and dedupe savings. Available CPU cycles in all-SSD systems can be used for running data reduction, both inline and in the background. Thus, savings in CPU cycles can directly benefit storage efficiency.

6.1 Reads: NVMe-oF Clients

In the first experiment, the load-generating clients used the NVMe-oF protocol (over a Fibre Channel network) to communicate with LUNs configured on ONTAP. Together with NVMe SSDs, when compared to earlier results, the latency and throughput numbers are both an order of magnitude better. At these low latencies, the experiment is more sensitive

[3] The SCSI specification does not require atomicity. ONTAP does not support WRITE_ATOMIC.

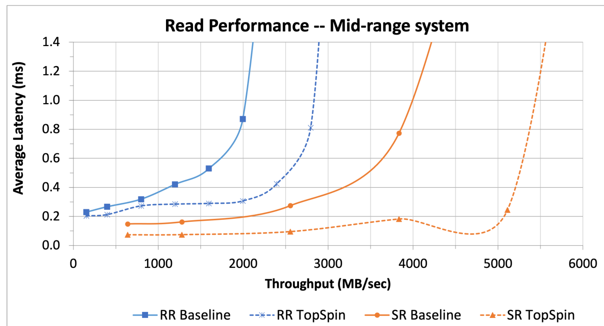


Figure 10: Latency vs achieved throughput on the 20-core system with increasing random read (RR) and sequential read (SR) load.

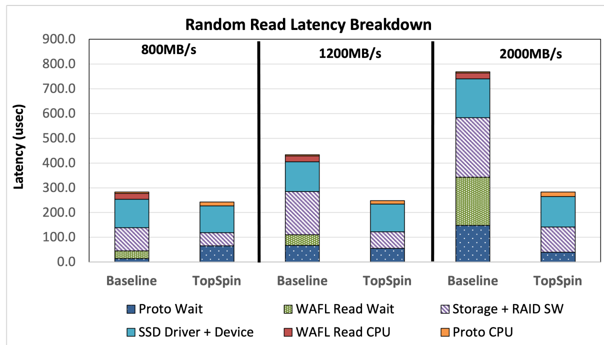


Figure 11: Latency across ONTAP components with and without TopSpin at three specific loads of random reads.

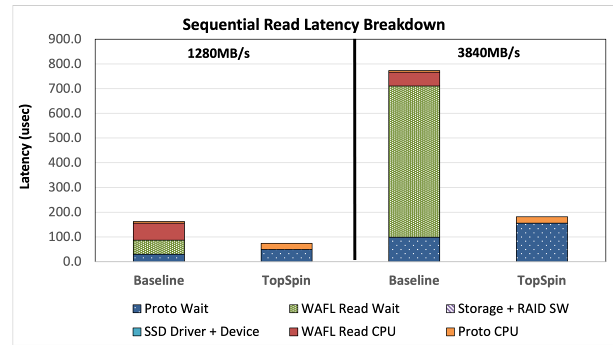


Figure 12: Latency across ONTAP components with and without TopSpin at two specific loads of sequential reads.

to software delays in ONTAP. Fig. 10 presents the latency vs achieved throughput on the mid-range 20-core controller with 8KiB random read (*RR*) [4] and 64KiB sequential read (*SR*) workloads. *Baseline* now includes the Fastpaths.

6.1.1 Random Read Performance

TopSpin shifts the curve to the right, for example doubling throughput at 400us latency. Customers can also operate their systems for higher throughput, with a tolerance for higher latency (e.g., 5ms). The *peak* throughput of a system is that achieved as the system approaches saturation and beyond which latencies grow exponentially. As with the Fastpaths, TopSpin delivers a 27% higher peak throughput because the streamlined I/O path reduces the CPU costs per operation (3.0GiB/s at 2.9ms latency vs. 2.4GiB/s at 3.2ms). In this test, TopSpin finds the required data in memory in 1.9% of reads, issues storage I/O directly in 97.7% of reads, and falls through to WAFL in only 0.4% of reads for reasons such as an unavailable SLC entry. Fig. 11 shows the per-component latency at three loads, and WAFL and Protocol latencies are now further divided into CPU time vs wait time. The data at low load (800MiB/s) approximates the breakdown for a single outstanding I/O. As load increases, the WAFL read message wait time becomes a significant factor (195us at 2GiB/s), and TopSpin eliminates it. Reduction in CPU consumption also yields lower wait times in other components. As we are now evaluating with NVMe SSDs, *SSD Driver+Device* latency ranges from 115us to 160us without TopSpin, compared to older generation SAS SSDs from Fig. 7. With TopSpin, this drops to 108us to 123us, due to decreases in driver scheduling delay. TopSpin replaces 24us of WAFL read message CPU time with a 10us increase in the *Proto CPU* time, which results in CPU cost per read dropping from 66us down to 52us. At 2GiB/s, the non-device related time drops from 690us to 160us, in better proportion with the 123us *SSD Driver+Device*. *With TopSpin, device latency is now the largest single component and non-device latencies remain below 60% of the total.*

[4] By this time, the official SPC-1 as well as our internal benchmarks had switched the I/O size from 4KiB to 8KiB for random I/O workloads.

6.1.2 Sequential Read Performance

In general, ONTAP is capable of much higher sequential read throughput because it uses speculative readahead to prefetch required data into memory. This effectively eliminates *SSD Driver+Device* from the latency path, as shown in Fig. 12. As expected, TopSpin eliminates *WAFL Read Wait* and replaces substantial *WAFL Read CPU* time (56us at 3840MiB/s) with a small increase in *Proto CPU* (20us), resulting in significant increase in throughput—e.g., a 40% increase at 800us latency. Peak throughput goes up by 19% (6.0GiB/s at 3.0ms vs. 5.0GiB/s at 3.2ms), due to a reduction in the per operation CPU cost from 251us to 209us. Thanks to readahead, TopSpin finds data in memory in over 99.99% of reads and avoids failing through to WAFL.

Unlike TopSpin random reads, readahead prefetching instantiates and inserts WAFL buffers. We next optimized the readahead engine to use iobuffers. This carries two benefits: (1) iobuffers are lighter-weight because they maintain less state and so access fewer cache lines, which reduces the CPU cost of processing both their insertion and eventual eviction. At 5.1GiB/s load, the average CPU cost

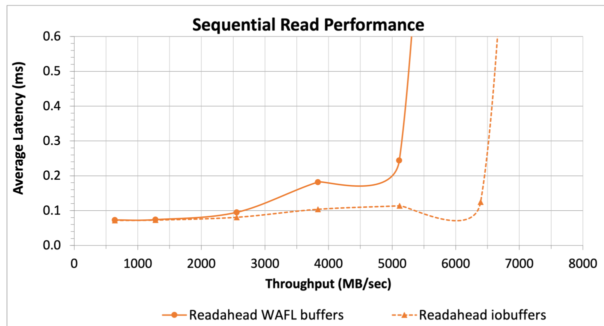


Figure 13: Latency vs achieved throughput on the 20-core system with TopSpin on sequential read load, using WAFL buffers and iobuffers for readhead.

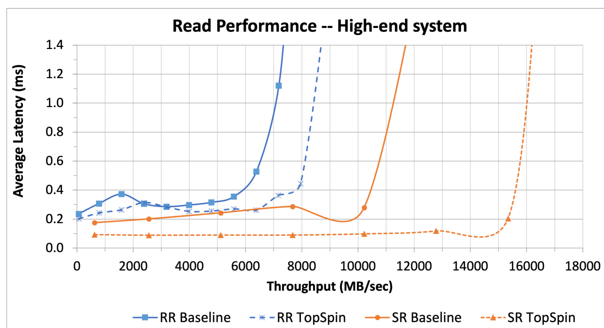


Figure 14: Latency vs achieved throughput on the 64-core system with increasing random read (RR) and sequential read (SR) load.

of readahead drops from 24.5% of all cores to 16.5%, and buffer scavenging drops from 15.0% of all cores to 6.0%. (2) iobuffers can be scavenged from outside of the WAFL component, which helps reduce overall WAFL wait times; none of that 6.0% scavenging CPU cost is in the WAFL component. Fig. 13 shows the results of this approach, including a 19% increase in peak throughput (7.2GiB/s at 2.3ms vs. 6.0GiB/s at 3.0ms).

6.1.3 High-end Read Performance

Fig. 14 reports the results of the same random and sequential read experiments on the high-end 64-core controller to study TopSpin on controllers with more CPU cores. Compared to the 20-core system, TopSpin benefits are similar for SR but are somewhat lower for RR. This shows that TopSpin benefits are greater for certain workloads when CPU resources are more limited. The latency bump around 2GiB/s for both RR graphs is due to the time-lag to activate the optimal number of threads in the (NVMe) Protocol component on this high-end controller; as mentioned earlier, ONTAP dynamically scales this number. The per-component latency breakdown (not shown) matches that of the 20-core controller.

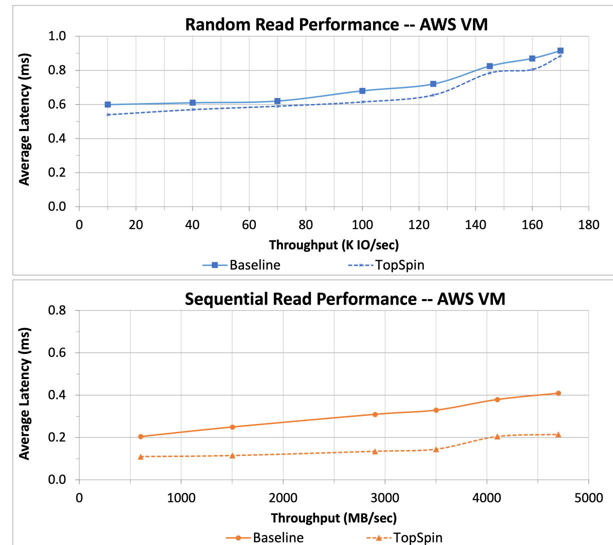


Figure 15: Latency vs achieved throughput on a VM in AWS with increasing random read (top) and sequential read (bottom) load.

6.2 Read Performance in Cloud Deployments

We next deployed a VM in the AWS public cloud containing 128 cores and 512GiB of DRAM, and used iSCSI clients to send a load of 8KiB random reads and 64KiB sequential reads, using a 2:1 compressible dataset. We attached io1 EBS [1] volumes to the VM over the network, exposed as NVMe SSD drives, with an EC2 entitlement of 160K ops/sec. We experimented with ONTAP 9.14.0, in which TopSpin was enabled for cloud VMs. Fig. 15 presents the measurements of server-side latencies and throughput. For random read, an abundance of CPU cores reduces internal queuing times and the benefits of TopSpin, and latency improvements range between 30us and 65us. In contrast, sequential read leverages readahead to hide the drive access times, and TopSpin nearly halves latencies at all loads. TopSpin-enabled cloud deployments will be available using Amazon FSx for NetApp ONTAP (FSxN [2]) later in 2024.

6.3 Mixed Read and Write Workloads

To measure the impacts of TopSpin on mixed read-write workloads and mixed random-sequential workloads, we ran the internal SPC-1 macrobenchmark on the 64-core controller, with clients connected to ONTAP using FCP. SPC-1 issues 40% reads and 60% writes, of which each are 40% sequential and 60% random [17]. These results are shown in Fig. 16, where *Write Baseline* and *Write TopSpin* are the observed write latency without and with TopSpin read enabled, respectively. In this case, overall throughput is not changed through the use of TopSpin, but the same peak throughput is achieved with 4.9% lower CPU usage. Read latency at peak throughput dropped 67%, from 442us to 147us. Further, with

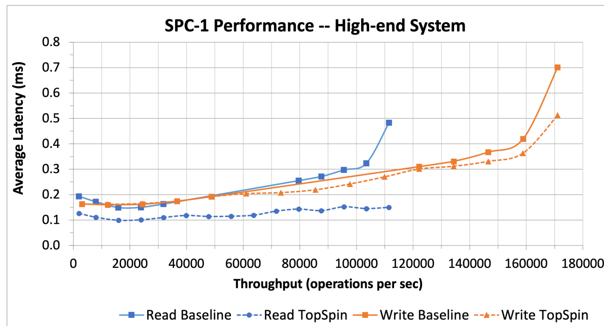


Figure 16: Read and write latencies vs achieved throughput on the 64-core system with increasing SPC-1 load.

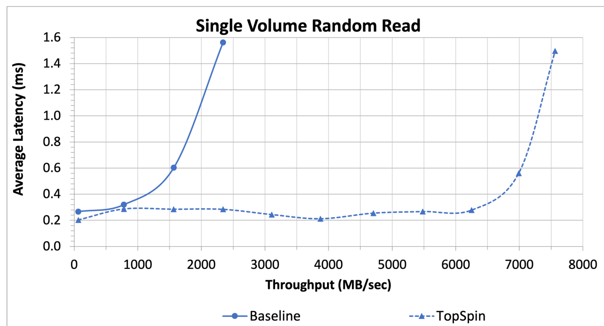


Figure 17: Latency vs achieved throughput on the 64-core system with increasing 8KiB random read load to a single volume.

the majority of reads now bypassing the WAFL component, write latency dropped from 637us down to 517us because wait time for WAFL write messages dropped from 350us to 189us. In this test, 82.4% of reads hit in the cache, 9.2% successfully read from storage, and 8.5% failed through to WAFL due to missing L1s or dirty L0s.

We next evaluated load to an Oracle database. Clients connected over FCP to a 2x18-core controller with 512GiB of DRAM. We compared ONTAP 9.2 to ONTAP 9.3, the first release with TopSpin. Load was generated to an Oracle 12c database using SLOB2 [9], comprising 75% SELECT and 25% UPDATE SQL commands. Peak throughput from the storage server increased from 345K I/Os per second to 400K I/Os per second. As explained earlier, WAFL is designed to complete writes quickly, so UPDATEs do not impact user sessions much. However, storage read latencies directly impact SELECTs, which dropped from 1.13ms to 0.95ms as reported by the database server.

6.4 Additional Benefits to Bypassing WAFL

Beyond the benefits already discussed, TopSpin also provides an effective way to work around a long-standing bottleneck in WAFL parallelism. The Waffinity model translates the WAFL file system into a static hierarchy of data partitions [12], with a single active thread per partition. However, the fixed number of partitions at each level of the hierarchy cannot guarantee optimal performance across all workloads. Although rarely encountered, Waffinity-unfriendly workloads are limited by the data partitioning to using a subset of the available cores. Some examples: the entire system load is to a single volume, all load is to a single LUN or file, or sequential read streams to a single LUN or file where consecutive I/Os move lockstep one partition at a time. Dynamically changing the number of data partitions based on observing the current workload is feasible, but has significant technical challenges. By avoiding the WAFL message, TopSpin parallelism for such a workload is limited only by the number of Protocol component threads. To evaluate one such case, we issued an 8KiB random read load from NVMe-oF clients directed to a single volume on the 2x32-core controller. Fig. 17 shows the results. Waffinity has only 9 client-facing data partitions per volume for WAFL read messages, so the baseline system saturates early once WAFL has utilized 9 cores. With TopSpin enabled, ONTAP activates more Protocol threads to use up to 31.5 cores for processing read operations. Combined with lower processing costs and fewer queuing delays, the increased parallelism yields 226.7% higher peak throughput, and even higher gains under 0.4ms. While this is an extreme case, TopSpin improves many similar scenarios with limited WAFL parallelism.

This section provides further evidence of the value of bypassing layers in ONTAP. It also encourages the continued adoption of TopSpin in other code paths. TopSpin has allowed us to incrementally achieve device-proportional overheads without requiring clean-sheet designs.

7 Lessons Learned

Lesson 1: *Bypassing layers for the error-free data path is an effective and safe way to eliminate software overhead in a modular system.* This approach retains useful component divisions and fails through to the component itself for special cases (only the error-free path needs to be optimized). Fail through correctness requires that such cases disregard any changes to message and system state caused by the partial Fastpath execution. Each successful optimization fueled the next project, and continues to do so. The optimization of other file system operations and protocols using TopSpin are in various stages of development and the design presented for reads has provided a strong foundation for these.

Lesson 2: *Incremental optimization for SSD was the right approach for ONTAP*. Before the Fastpath work, it was widely assumed that ONTAP would not be able to achieve device-proportional software overhead. NetApp thus developed and productized the alpha version of a clean-sheet design SSD-optimized storage system called FlashRay. FlashRay was discontinued for two primary reasons: (a) the roadmap to achieve feature-parity with ONTAP was multiple years and (b) the success of the Fastpaths demonstrated that

ONTAP could be optimized to achieve SSD-proportional latencies. Our incremental approach enabled NetApp to productize all-SSD ONTAP systems that were competitive on price and performance, while preserving legacy features. Critically, the WAFL file system architecture was already well-suited for SSD properties. In our experience, building a fast I/O path was significantly easier than building an entirely new file system with a rich feature set.

These lessons are applicable to other legacy systems and can influence designs of storage systems for new media. As new and faster media become available, future systems will need to go further in lowering software overhead. The remaining non-error message hops will need to be eliminated, such as special cases in TopSpin and even for device access. Subsequently, all I/O code paths will need to be further analyzed (such as for cache line misses) and optimized.

8 Related Work

I/O path optimization for low-latency SSD drives is an area of substantial study, notably bringing software overhead in proportion with device latencies. Shin, et al. [57] eliminate interrupt bottom halves and queue running contexts in the I/O completion path. BarrierFS [65] reduces software overhead by replacing expensive storage device I/O order guarantee approaches. ReFlex [35] builds a highly-optimized, run-to-completion execution model for remote NVMe Flash storage on top of the IX dataplane OS [4]. With only 21us software overhead, ReFlex is fast.

Kernel bypass is another popular approach. NVMeDirect [33] allows user-space applications direct access to the I/O device. Demikernel [69] is a datapath OS that uses kernel bypass devices and an optimized core scheduler for microsecond-scale latencies, even while retaining critical OS functionality. XRP [71] allows the user to embed application logic within the device driver's interrupt handler using eBPF. These hooks include file system state that can traverse on-disk structures and initiate new I/Os without returning control back to the application. These approaches are largely orthogonal to our work because the components of ONTAP discussed in this paper all run inside the kernel.

Techniques to reduce software overheads for low-latency I/O devices include RAID optimizations [63], CPU-scalable drive access [45], transparent zero-copy [59], and overlapping processing with device access [40]. i10 [21] provides a CPU-efficient RDMA remote storage stack, which minimizes the number of cores required to saturate both network and storage devices. SpanFS [24] partitions the file system into independent micro-services by file and directory to increase parallelism of the storage software, which is somewhat similar to Waffinity. Blk-switch [22] treats the storage stack like a network switch, and adapts networking optimizations to minimize software overhead and maximize drive throughput. Many systems optimize for predictable latency from SSD drives [5, 20, 62, 70, 34, 31, 61, 43, 56, 58, 22, 35], some using machine learning [18]. Fast core scheduling can provide QoS at microsecond granularity to latency-sensitive applications [49, 16].

Previous work analyzed low-latency drive performance [36] and its impact on the Linux storage stack [53]. Oh, et al. [48] optimize Ceph to adapt from HDDs to SSDs. I/O schedulers have been optimized [68] for low-latency devices and even evaluated as software overhead [64]. Performance requirements of Key-Value stores have inspired optimizations for these drives, including optimized CPU usage [41, 37, 42] and CPU bypass [46]. Lastly, persistent memory technologies place even more emphasis on low processing costs [6, 66, 39, 23], kernel bypass [8, 7], and indexing overheads [39, 23, 47, 44].

Our work was done on a 30+ year old legacy system without compromising the dozens of enterprise quality features that are critical to our customers. The interactions of our improvements with these features created additional challenges in our designs and implementations. We achieved significant performance gains while retaining the existing behavior of millions of lines of WAFL and ONTAP code outside the read path, despite potentially operating on the same data.

9 Conclusion

Although several aspects of the decades-old ONTAP architecture were well-suited for building an all-SSD controller, the software delays (proportional to HDD latencies) in its legacy I/O path had made that impractical. In this paper, we presented the multi-year journey of incremental improvements to the read path that have reigned in software overhead and made the all-SSD ONTAP controller a success. In future work, we plan to present data reduction technologies that were also crucial to this productization, as well as extensions of TopSpin to other operations, such as writes.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Angelos Bilas, whose suggestions have significantly improved this paper. This paper discusses the results of a decade of engineering effort by a huge group of talented engineers: Abdul Basit, Joseph Brown, Yong Cho, Roopesh Chuggani, Peter Denz, Ravi Dronamraju, Manish Katiyar, Rajesh Khandelwal, Aditya Kulkarni, Szu-wen Kuo, Asif Pathan, James Pitcairn-Hill, Parag Sarfare, Girish Hebbale Venkata Subbaiah, Ananthan Subramanian, Venkateswarlu Tella, Daniel Ting, Sriram Venketaraman, Jungsook Yang, Xiaoyan Yang, and many others.

References

[1] Amazon. Amazon elastic block store. https://aws.amazon.com/ebs/.

[2] Amazon. What is Amazon FSx for NetApp ONTAP? https://docs.aws.amazon.com/fsx/latest/ONTAPGuide/what-is-fsx-ontap.html.

[3] Wendy Bartlett and Lisa Spainhower. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on dependable and secure computing*, 1(1), 2004.

[4] 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)*, October 2014.

[5] Matias Bjørling, Javier González, and Philippe Bonnet. LightNVM: The linux open-channel SSD subsystem. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2017.

[6] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, 2012.

[7] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. Scalable persistent memory file system with Kernel-Userspace collaboration. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, February 2021.

[8] Jungsik Choi, Jiwon Kim, and Hwansoo Han. Efficient memory mapped file I/O for In-Memory file systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, July 2017.

[9] Kevin Closson. SLOB resources. https://kevinclosson.net/slob/.

[10] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-Diagonal parity for double disk failure correction. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*, March 2004.

[11] Storage Performance Council. Storage performance council-1 benchmark. www.storageperformance.org.

[12] Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni. To waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceeding of Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[13] Matthew Curtis-Maury, Ram Kesavan, and Mrinal Bhattacharjee. Scalable write allocation in the WAFL file system. In *Proceedings of the Internal Conference on Parallel Processing (ICPP)*, 2017.

[14] Peter Denz, Matthew Curtis-Maury, and Vinay Devadas. Think global, act local: A buffer cache design for global ordering and parallel processing in the WAFL file system. In *Proceedings of the Internal Conference on Parallel Processing (ICPP)*, 2016.

[15] John K Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A Smith, et al. FlexVol: flexible, efficient file volume virtualization in WAFL. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2008.

[16] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, November 2020.

[17] Binny Gill. SPC-1 benchmark. https://www.usenix.org/legacy/events/fast05/tech/full_papers/gill/gill_html/node28.html.

[18] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on unpredictable flash storage with a light neural network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, November 2020.

[19] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of USENIX Winter Technical Conference*, 1994.

[20] 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)*, February 2017.

[21] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP=RDMA: CPU-efficient remote storage access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, February 2020.

[22] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting linux storage stack for µs latency and high throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 113–128. USENIX Association, July 2021.

[23] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the Symposium on Operating System Principles (SOSP)*, SOSP '19, 2019.

[24] 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 (USENIX ATC 15)*, July 2015.

[25] Ram Kesavan, Matthew Curtis-Maury, and Mrinal Bhattacharjee. Efficient search for free blocks in the WAFL file system. In *Proceedings of the Internal Conference on Parallel Processing (ICPP)*, 2018.

[26] Ram Kesavan, Matthew Curtis-Maury, Vinay Devadas, and Kesari Mishra. Storage gardening: Using a virtualization layer for efficient defragmentation in the wafl file system. In *17th Usenix Conference on File and Storage Technologies (FAST)*, 2019.

[27] Ram Kesavan, Matthew Curtis-Maury, Vinay Devadas, and Kesari Mishra. Countering fragmentation in an enterprise storage system. *ACM Transactions on Storage*, 15(4), jan 2020.

[28] Ram Kesavan, Harendra Kumar, and Sushrut Bhowmick. Wafl iron: Repairing live enterprise file systems. In *16th Usenix Conference on File and Storage Technologies (FAST)*, 2018.

[29] Ram Kesavan, Rohit Singh, Travis Grusecki, and Yuvraj Patel. Algorithms and data structures for efficient free space reclamation in WAFL. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2017.

[30] Ram Kesavan, Rohit Singh, Travis Grusecki, and Yuvraj Patel. Efficient free space reclamation in WAFL. *ACM Transactions on Storage (ToS)*, 13, October 2017.

[31] Bryan S. Kim, Hyun Suk Yang, and Sang Lyul Min. AutoSSD: an autonomic SSD architecture. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, July 2018.

[32] Byungseok Kim, Jaeho Kim, and Sam H. Noh. Managing array of SSDs when the storage device is no longer the performance bottleneck. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, July 2017.

[33] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, June 2016.

[34] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. Enlightening the I/O path: A holistic approach for application performance. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, February 2017.

[35] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash = local flash. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS 18)*, ASPLOS '17, April 2017.

[36] Sungjoon Koh, Changrim Lee, Miryeong Kwon, and Myoungsoo Jung. Exploring system challenges of Ultra-Low latency solid state drives. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, July 2018.

[37] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast NVM storage with uDepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, February 2019.

[38] 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)*, 2017.

[39] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the Symposium on Operating System Principles (SOSP)*, SOSP '17, 2017.

[40] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. Asynchronous I/O stack: A low-latency kernel I/O stack for Ultra-Low latency SSDs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, July 2019.

[41] 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 Symposium on Operating System Principles (SOSP)*, SOSP '19, 2019.

[42] Haoyu Li, Sheng Jiang, Chen Chen, Ashwini Raina, Xingyu Zhu, Changxu Luo, and Asaf Cidon. RubbleDB: CPU-Efficient replication with NVMe-oF. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, July 2023.

[43] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. Ioda: A host/device co-design for strong predictability contract on modern flash storage. In *Proceedings of the Symposium on Operating System Principles (SOSP)*, SOSP '21, 2021.

[44] Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, and Ding Yuan. ctFS: Replacing file indexing with hardware memory translation through contiguous file allocation for persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, February 2022.

[45] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Max: A Multicore-Accelerated file system for flash storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, July 2021.

[46] 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 13)*, June 2013.

[47] Ian Neal, Gefei Zuo, Eric Shiple, Tanvir Ahmed Khan, Youngjin Kwon, Simon Peter, and Baris Kasikci. Rethinking file mapping for persistent memory. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, February 2021.

[48] Myoungwon Oh, Jugwan Eom, Jungyeon Yoon, Jae Yeun Yun, Seungmin Kim, and Heon Y. Yeom. Performance optimization for all flash scale-out storage. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 2016.

[49] 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)*, February 2019.

[50] Naresh M. Patel. Half-latency rule for finding the knee of the latency curve. *SIGMETRICS Perform. Eval. Rev.*, 43(2), sep 2015.

[51] Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. SnapMirror: File-system-based asynchronous mirroring for disaster recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*. USENIX Association, 2002.

[52] Samsung. Pm1725 nvme pcie ssd. `https://www.samsung.com/us/labs/pdfs/collateral/pm1725-ProdOverview-2015.pdf`.

[53] Eric Seppanen, Matthew T. O'Keefe, and David J. Lilja. High performance solid state storage under linux. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, 2010.

[54] Skip Shapiro. Technical report: Flash cache best practice guide. `https://www.netapp.com/pdf.html?item=/media/19754-tr-3832.pdf`.

[55] Skip Shapiro. Technical report: Flash pool design and implementation guide. `https://www.netapp.com/pdf.html?item=/media/19681-tr-4070.pdf`.

[56] Kai Shen and Stan Park. FlashFQ: A fair queueing I/O scheduler for Flash-Based SSDs. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, June 2013.

[57] Woong Shin, Qichen Chen, Myoungwon Oh, Hyeonsang Eom, and Heon Y. Yeom. OS I/O path optimizations for flash solid-state drives. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, June 2014.

[58] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. Flash on rails: Consistent flash performance through redundancy. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, June 2014.

[59] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. zIO: Accelerating IO-Intensive applications with transparent Zero-Copy IO. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, July 2022.

[60] Rajesh Sundaram. The Private Lives of Disk Drives. https://www.netapp.com/atg/publications/publications-the-private-lives-of-disk-drives-20064017/, 2006.

[61] Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. Optimizing storage performance with calibrated interrupts. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, July 2021.

[62] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika MansouriGhiasi, Lois Orosa, Juan Gomez-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)*, 2018.

[63] Shucheng Wang, Qiang Cao, Ziyi Lu, Hong Jiang, Jie Yao, and Yuanyuan Dong. StRAID: Stripe-threaded architecture for parity-based RAIDs with ultra-fast SSDs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, July 2022.

[64] Caeden Whitaker, Sidharth Sundar, Bryan Harris, and Nihat Altiparmak. Do we still need io schedulers for low-latency disks? In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 23)*, HotStorage '23, 2023.

[65] 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 18)*, February 2018.

[66] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2016.

[67] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't stack your log on my log. In *INFLOW*, 2014.

[68] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Split-level i/o scheduling. In *Proceedings of the Symposium on Operating System Principles (SOSP)*, SOSP '15, 2015.

[69] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195?211, New York, NY, USA, 2021. Association for Computing Machinery.

[70] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. 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)*, October 2018.

[71] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, July 2022.

NETAPP, the NETAPP logo, and the marks listed at http://www.netapp.com/TM are trademarks of NetApp, Inc. Other company and product names may be trademarks of their respective owners.

We Ain't Afraid of No File Fragmentation: Causes and Prevention of Its Performance Impact on Modern Flash SSDs

Yuhun Jun[1,2], Shinhyun Park[3], Jeong-Uk Kang[2], Sang-Hoon Kim[4] and Euiseong Seo[*3]
[1] *Department of Semiconductor and Display Engineering, Sungkyunkwan University*
[2] *Memory Business Unit, Samsung Electronics Co. Ltd.*
[3] *Department of Computer Science and Engineering, Sungkyunkwan University*
[4] *Department of Software and Computer Engineering, Ajou University*
[*] *Corresponding Author: Euiseong Seo (euiseong@skku.edu)*

Abstract

A few studies reported that fragmentation still adversely affects the performance of flash solid-state disks (SSDs) particularly through request splitting. This research investigates the fragmentation-induced performance degradation across three levels: kernel I/O path, host-storage interface, and flash memory accesses in SSDs. Our analysis reveals that, contrary to assertions in existing literature, the primary cause of the degraded performance is not due to request splitting but stems from a significant increase in die-level collisions. In SSDs, when other writes come between writes of neighboring file blocks, the file blocks are not placed on consecutive dies, resulting in random die allocation. This randomness escalates the chances of die-level collisions, causing deteriorated read performance later. We also reveal that this may happen when a file is overwritten. To counteract this, we propose an NVMe command extension combined with a page-to-die allocation algorithm designed to ensure that contiguous blocks always land on successive dies, even in the face of file fragmentation or overwrites. Evaluations with commercial SSDs and an SSD emulator indicate that our approach effectively curtails the read performance drop arising from both fragmentation and overwrites, all without the need for defragmentation. Representatively, when a 162 MB SQLite database was fragmented into 10,011 pieces, our approach limited the performance drop to 3.5%, while the conventional system experienced a 40% decline.

1 Introduction

File system fragmentation, in which discontinuities exist between data blocks belonging to a single file, transforms sequential access to the file into a series of random accesses to scattered chunks at the storage level [35, 37]. In the era of hard disks (HDDs), which suffer from considerably long seek delays for random accesses, this resulted in additional seek operations and ended up with significantly impaired read performance [7].

To prevent performance degradation caused by fragmentation, file systems utilize various techniques [35], such as delayed allocation [23] and preallocation of data blocks [2], to maintain continuity among data blocks. Nonetheless, it is inherently challenging to avoid situations where the file system cannot locate free data blocks immediately adjacent to a file's data blocks, either due to the simultaneous writing of multiple files or appending to a file after a significant amount of time has passed since its last write.

In contrast to HDDs, flash-based solid-state disks (SSDs) eliminate mechanical movements, significantly reducing the performance gap between random and sequential accesses. However, recent studies have revealed that SSDs also experience a two to five times slower read performance when accessing fragmented files [4], prompting the development of several defragmentation schemes to address this performance decline [13, 31, 42]. However, these studies only superficially observed the performance degradation based on the fragmentation patterns and hypothesized that its primary cause is *request splitting* in the kernel I/O path due to fragmentation [13, 31].

In this paper, through a series of experiments, we reveal that the previous claim suggesting file fragmentation adversely impacts sequential read performance also in flash SSDs due to request splitting is based on inaccurate experiment settings and analyses. Moreover, we demonstrate that during file fragmentation, the page-to-die mappings within the SSD deviate from the ideal state, leading to a substantially increased number of *die-level collisions* [18] compared to the cases without file fragmentation. This increase in die-level collisions, which leads to the degradation of SSD's internal parallelism, is the primary contributing factor to the observed deterioration in read performance in an SSD with file fragmentation.

An SSD's firmware allocates its flash memory pages in a round-robin manner across the flash memory dies based on the order in which they are written. Consequently, in situations where file fragmentation occurs, the pages storing contiguous file blocks cannot be placed on contiguous dies but are instead allocated to arbitrary dies. To prevent such improper

page-to-die mapping patterns arising from file fragmentation, we propose a simple extension to the NVMe protocol that provides hints for page-to-die mapping in conjunction with a write command. With these hints, the page for an appending write is mapped to the die following the die where the previous file block's page was assigned to. In addition, the page for an overwrite operation to an existing file block, which also disrupts the page-to-die mapping pattern, is mapped to the same die where the original page was located. Through these simple hints and mapping rules, it is possible to avoid performance degradation in read operations even in situations with file fragmentation or overwrites to existing files. We evaluate the proposed approach using two configurations: first, through emulation with commercial SSDs, and second, by implementing it in the Linux kernel and NVMeVirt [22], an SSD emulator.

To the best of our knowledge, this research is the first to experimentally demonstrate that the primary cause of file fragmentation-induced performance degradation in an SSD is the deterioration of its internal parallelism. Moreover, we show that this performance degradation is not an inevitable consequence of fragmentation and can be easily avoided while keeping the fragmentation state unchanged.

The rest of this paper is organized as follows. After introducing the background and related work on the file system fragmentation in Section 2, Section 3 analyzes its impact on performance when using flash SSDs. Section 4 proposes our approach to avoid performance degradation from file fragmentation and overwrite operations, and Section 5 evaluates the proposed approach. Finally, Section 6 concludes the research.

2 Background and Motivation

2.1 Old Wisdom on File Fragmentation

In the HDD era, the primary and direct cause of performance degradation from file fragmentation was the seek time between dispersed sectors of the file [7]. File fragmentation has a more pronounced negative impact on read operations, which must wait for the completion, compared to writes that can be buffered by the storage. The long seek time of HDDs overshadowed other factors that negatively impacted performance due to file fragmentation. However, file fragmentation adversely affects performance at three levels: kernel I/O path, storage device interface, and storage media access.

As shown in Fig. 1, to the file system, a file is an array of file blocks, which are logically contiguous. However, the file system data blocks where these file blocks are actually stored may not be contiguous. Naturally, file systems strive to store contiguous file's logical blocks in contiguous file system data blocks. However, it is difficult to achieve a completely fragmentation-free data block allocation, especially when a file grows incrementally over time, as other files may be written behind the last written data block. Therefore, the data blocks of a file can be allocated in separate locations.

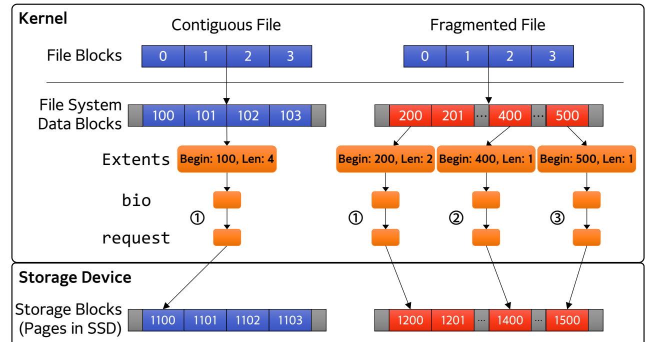


Figure 1: A sequential access to a contiguous file is translated to a single device command while that to a fragmented file ends up with multiple requests.

Only a single command is required for the host to instruct the storage device to perform read or write operations on contiguous storage space. Thus, when a sequential read occurs for a file, the Linux kernel reads the data block mapping in the file's inode, and for each contiguous data block region, it creates a `bio` (block I/O) data structure. This data structure is used to create the corresponding `request` data structure to be passed to the device driver, which then issues the command for the request to the device. Through this process, a single sequential file access may be split into multiple `bio`s and corresponding requests to the storage device, depending on the degree of file fragmentation.

This request splitting is known to increase I/O execution time, as it increases the number of data structure creations and calls to underlying functions, including the device driver code [13, 16, 17, 31, 32]. Naturally, the increased number of device commands leads to time delays at the SATA [34] or NVMe [9] interface level. The increased number of storage device commands leads to an increased time for the storage device's firmware to process them. Specifically, the frequency of fetching, decoding, translating commands into storage media operations, and queuing media access operations increases. Therefore, file fragmentation also delays the processing time of the storage device controller.

Finally, file fragmentation extends the time to access storage media in the storage device. As mentioned earlier, in the case of HDDs, seek time is required for the disk head to move to the track where the requested sector is located, and a disk rotation delay occurs to locate it on the track [7]. However, unlike performance degradation caused by the kernel I/O path and storage device interface, SSDs are expected to not experience an increase in the storage medium access time due to fragmentation, as SSDs do not have seek time and rotational delay [10, 12, 39, 41].

To address the fragmentation-induced performance decline, two types of studies have been conducted: one aims to prevent fragmentation from occurring, and the other aims to recover

file access performance by transforming fragmented files into contiguous ones.

The delayed allocation technique used in the ext4 file system performs data block allocation not at the write system call handling but at the time of page flush [23]. In cases where small write operations are interleaved with writes to other files, delayed allocation increases the likelihood that write operations for a file are allocated to contiguous data blocks.

In addition, ext4 reserves a predefined window of free data blocks for each file's inode. These reserved free blocks will be actually allocated to the file for its successive append writes. This significantly reduces the occurrences of fragmentation especially when multiple files in the same directory are simultaneously written [2, 35].

While these techniques can reduce the frequency of file fragmentation, research has shown that it is an inevitable result of file system aging [5, 37]. Therefore, various defragmentation tools have been proposed to rewrite scattered file data blocks to contiguous ones to recover the I/O performance [8, 25, 27, 30, 35].

Sato proposed an online defragmentation tool for the Linux ext4 file system [35]. The proposed scheme allocates contiguous free blocks to a temporary inode, copies the fragmented file data to the temporary inode, deletes the original file, and renames the temporary inode to the original's.

Various techniques have been proposed to mitigate the overhead caused by defragmentation, as copying all fragmented files can take a significant amount of time. For example, F2FS's defragmentation tool, `defrag.f2fs`, allows users to selectively migrate only the user-selected area by manually inputting the starting block address, length, and target location as parameters [30]. XFS's `xfs_fsr` sorts files by their number of extents and groups the top 10% of files into a unit called a *pass*, performing defragmentation for each pass [27]. Btrfs's built-in defragmentation tool defragments only extents smaller than the target extent size specified as a parameter [33]. However, ultimately, defragmentation consumes a significant amount of time and energy as it induces a large number of read and write operations on relatively slow storage devices [13, 31].

2.2 File Fragmentation in SSD-Era

Most researchers and SSD manufacturers initially claimed that SSD performance is not affected by file fragmentation, and that defragmentation is unnecessary and may even be harmful due to the write operations involved in the defragmentation process, which can reduce the lifespan of the flash memory [10, 12, 39, 41]. However, contrary to initial claims that SSDs do not have fragmentation issues, some researchers observed performance degradation due to file system aging and resulting fragmentation.

SSDs offer significantly higher performance than a single flash memory die (chip) because they operate multiple flash dies in parallel [21]. Specifically, NVMe SSDs offer 65,535 command queues, each capable of queueing 65,536 commands. Even when fragmentation leads to smaller request sizes that cannot fully utilize die-level parallelism, smaller flash operations in the command queues can still be processed out-of-order, allowing most dies to be fully utilized. This enables SSDs to achieve performance close to their maximum potential even when accessing small fragments. Consequently, some researchers speculated that the kernel I/O path and interface overhead due to request splitting have a greater impact on fragmentation-induced performance degradation than flash memory access time [13, 16, 17, 31, 32].

Conway et al. empirically observed performance degradation in various workloads due to file system aging on SSDs [4,5]. They discovered that file fragmentation frequently occurs on SSDs as the file system ages. In scenarios where the `git` pull commands are executed repeatedly, they observed that read performance can be degraded by up to five times. Geriatrix is a tool capable of effectively emulating file system and storage aging [20]. Using Geriatrix, the authors demonstrated a performance degradation of up to 78% due to file system aging on SSDs. While both studies observed changes in file system layout and performance degradation due to file system aging in various circumstances, they did not conduct an in-depth analysis of the underlying causes for this performance decline.

Park and Eom argued that the main cause of performance degradation due to fragmentation on SSDs is request splitting, and thus the distance between data blocks does not significantly affect read performance, while the degree of fragmentation does [31]. In a subsequent paper, they made a contradictory claim, stating that the distance between fragmented blocks also significantly affects performance on SSDs [32]. In addition, they proposed FragPicker, an efficient defragmentation approach that carries out online migration of fragments only that have been accessed [31].

Zhu et al. proposed a scheme that can simultaneously issue parallel I/O requests for defragmentation in ext4, minimizing defragmentation time and maximizing SSD internal parallelism. This approach improved defragmentation time by three times compared to the traditional `e4defrag` [42].

Regarding these conflicting claims, we clarify in Section 3 that this arises because previous studies' experimental setups fail to distinguish between performance degradation directly caused by fragmentation and that indirectly caused by the influence of fragmentation on SSD's internal data placement.

2.3 Internals of Modern Flash SSDs

As previously mentioned, a modern flash SSD is equipped with multiple flash dies that can operate in parallel. To maximize the bandwidth and throughput of an SSD, it is crucial to maintain a high degree of die-level parallelism [3, 21]. Conversely, since a die can only process one request at a

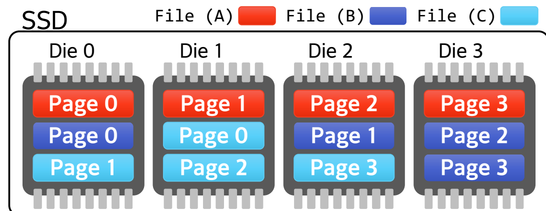


Figure 2: Data placement of three files in a flash SSD where one is contiguous and the other two are fragmented.

time [14, 40], if the pages to be read are stored on a single die, the read requests for these pages must be serialized within that die. This die-level read collision significantly degrades read performance [18].

To prevent die-level collisions for read operations, the flash translation layer (FTL) of an SSD's firmware must perform physical page allocation in a manner that distributes the physical pages storing contiguous logical pages across as many dies as possible. For this purpose, the FTL of most modern SSDs selects a die in a round-robin manner when allocating a flash page for processing an incoming page write request [3, 19]. Additionally, modern FTLs perform the valid page copy within the die where the page resides during the garbage collection (GC) process if the die has a sufficient number of free pages [11]. This allows for the maintenance of die parallelism. However, since GC occurs in parallel across all dies, this strategy does not significantly impact the performance of GC.

For example, in Fig. 2, File A is evenly distributed across four dies since its four pages were written without interference. Thus, a sequential read of File A will be performed simultaneously on these four dies, resulting in a bandwidth of up to four times the flash die performance. In contrast, assume that the writes to File B and File C were interleaved. As the die for storing a logical page is assigned in a round-robin manner according to the order of writes performed within the SSD, both the third and last pages of File B ended up being allocated to Die 3. As a result, the time to read File B is twice as long as that for reading an ideally-placed file of the same size, such as File A.

File fragmentation occurs in most cases when multiple files are simultaneously written [4, 31]. Therefore, when file fragmentation occurs, the die allocation of flash pages associated with a file might not be evenly distributed, leading to the pages of a single file being consolidated on certain dies. This phenomenon arises because the FTL allocates dies for pages solely based on their incoming order. However, the presence of file fragmentation does not inevitably result in uneven page distribution over dies, just as a contiguous file does not guarantee that its pages will always be evenly and sequentially allocated on consecutive dies.

Through ext4's preallocation, data blocks can be allocated contiguously in the file system, even if writes from other files occur in between. However, since the die mapping of flash pages takes place *at the actual moment of their writing inside the SSD*, even files that are contiguous at the file system level can exhibit uneven page distribution in the SSD. Conversely, if the data blocks of a fragmented file are written at the appropriate timing, it is possible for the file's pages to be distributed evenly across all dies.

Table 1: System configurations for experiments.

Processor	Intel Xeon Gold 6138 2.0 GHz, 160-Core
Chipset	Intel C621
Memory	DDR4 2666 MHz, 32 GB x16
OS	Ubuntu 20.04 Server (kernel v5.15.0)
Interface	PCIe Gen 3 x4 and SATA 3.0
Storage	NVMe-A: Samsung 980 PRO 1 TB NVMe-B: WD Black SN850 1 TB NVMe-C: SK Hynix Platinum P41 1 TB NVMe-D: Crucial P5 Plus 1 TB
	SATA-A: Samsung 870 EVO 500 GB SATA-B: WD Blue SA510 500 GB

In addition to the fragmentation cases, irregular die allocation may occur in cases of file overwrites. Assume a file stored in contiguous file system blocks has its pages sequentially allocated to dies on the SSD. In this ideal situation, if an overwrite is performed on a middle block of the file, the SSD must allocate a new page for that block and invalidate the page currently mapped to the block due to the nature of flash memory, which does not allow in-place updates. At this point, the new page will be allocated from the die next to the one that last allocated a page, according to the round-robin policy. As a result, there is a high likelihood that the new page will not be located on the same die as the original page, leading to a considerable decline in performance due to die-level collisions when conducting a sequential read on the file.

3 Analysis of File Fragmentation

This section explores the cause behind fragmentation-induced performance degradation through a series of experiments. The configuration of the experimental system for our analysis is described in Table 1. We used the ext4 file system [28]. To minimize the influence of the kernel's page cache and extent cache on the experimental results, we performed a cache drop before each experiment run. In addition, to adjust the block I/O request queue depth for our experiments, we used the kernel's `nr_requests` parameter. All experimental results are an average of 10 repetitions.

To begin, we examined the performance drop in ext4 on NVMe SSDs based on the degree of fragmentation (DoF), which is the ratio of the actual number of extents to the ideal number of extents [17]. For this, we created a set of files that have various DoF. Each fragmented file, with a size of 8 MB, is created by interleaving the writes to the target file and that to a dummy file as many times as the desired DoF. The size of

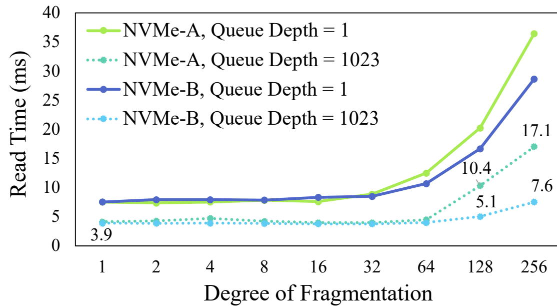


Figure 3: Time taken for reading an 8 MB file stored on NVMe SSDs while varying its DoF.

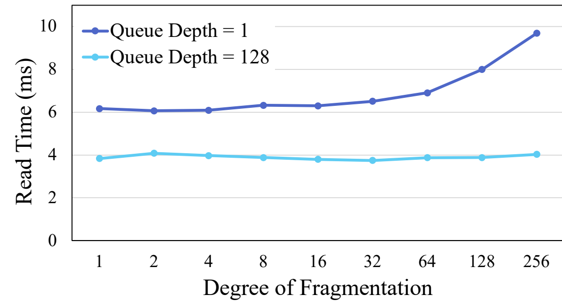


Figure 4: Time for sequentially reading an 8 MB file stored on `ramdisk` depending on its DoF.

the write to the dummy file between the writes to the target file was determined so that the offset between the two fragments of the target file becomes 8 MB. For example, if the target DoF is 4, four fragments or extents must compose the 8 MB target file. We wrote the first quarter of the target file first and then wrote 6 MB for the dummy file. By repeating this four times, we can obtain an 8MB file with a DoF of 4.

We varied the DoF in our analysis from 1, representing contiguous files, to 256, unlike previous studies that went beyond this range. A fragment size when the DoF is 256 in our analysis is 32 KB. Due to the aforementioned delayed allocation and block reservation techniques, which are used by ext4 to suppress fragmentation, it is highly unlikely for a fragment to have a smaller size than that.

In order to create a file exactly with the desired DoF using this method, it is necessary to disable delayed allocation and block reservation. To disable delayed allocation, we used the `direct` mode when writing files and provided the `nodelalloc` mount option. Block reservation was neutralized by setting both the `reserved_clusters` runtime parameter and the percentage of reserved block parameter, which is `mkfs`'s, to 0. We also disabled the per-inode preallocation feature. Every fragmented file was verified whether it has the desired DoF by using the `filefrag` tool [26]. A single extent in ext4 can represent 2^{15} contiguous blocks, or 128 MB, with a 4 KB file system block size [28]. Since the fragments were all smaller than 128 MB, the desired DoF value precisely matched the number of extents constituting the file.

As shown in Fig. 3, the read performance of both NVMe SSDs decreased from the point where the DoF exceeded 64, regardless of the request queue depth. Since the default maximum request size for the Linux kernel is 1 MB, no performance difference was observed when the fragment size exceeded 1 MB, corresponding to situations where the DoF is 8 or lower.

Notably, both SSDs showed a more drastic performance change when the queue depth was set to 1. When the I/O queue depth was set to 1023, which is the Linux default value for an NVMe SSD, the execution time was shorter, and the performance degradation due to fragmentation was less pronounced compared to when the queue depth was set to 1. However, even at a queue depth of 1023, the execution time increased significantly with the increase in DoF. NVMe-A exhibited 2.7 times and 4.4 times longer execution times at DoF 128 and 256, respectively, compared to DoF 1. Similarly, NVMe-B demonstrated 1.3 times and 1.9 times longer execution times at DoF 128 and 256, respectively.

In this experiment, we have confirmed that file fragmentation indeed causes performance degradation in SSDs. To further elucidate the specific causes of this performance degradation, subsequent experiments were conducted.

3.1 Impact Caused by Request Splitting

As previously mentioned, file fragmentation results in request splitting, where a single I/O operation is translated into multiple device commands. The impact of increased processing time at the host side due to request splitting on performance degradation will be more evident when the storage device's processing time is shorter. Therefore, we measured the delay occurring in the kernel I/O path due to request splitting by using a *ramdisk* as the storage device, which has an extremely short host-to-storage interface and storage media access times.

Fig. 4 shows the sequential read performance of files stored on the ramdisk according to their DoF. Since the ramdisk has extremely fast access speed and there is negligible difference between random and sequential access times, the performance changes observed in Fig. 4 can be attributed primarily to the difference in time consumed by the kernel I/O path rather than the storage media. Our experiments revealed that as the DoF increased with the request queue depth set to 1, read performance decreased, resulting in a 1.5-fold increase in read time when the DoF was 256.

As stated, one request to the storage device is generated for each contiguous storage address range. Consequently, the numbers of `iomap` structures, which are required for direct I/O, `bio` structures, and `request` structures increased with the DoF. Additionally, the number of function invocations for their creation also rose as the DoF grew. When the queue depth was set to 1, these procedures were performed syn-

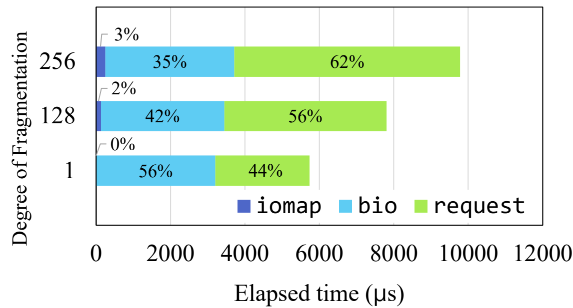


Figure 5: Time composition for creating request data structures in the kernel I/O path depending on File's DoF.

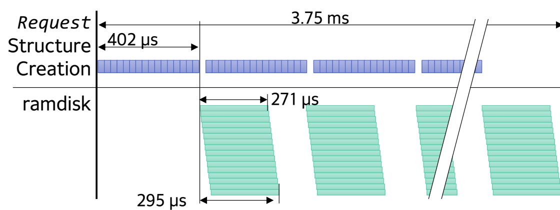


Figure 6: Reduction of read time due to the overlap of storage operations and `request` creation when File's DoF is 128.

chronously. As a result, the processing time for the kernel I/O path increased with the rise of DoF, and this increase became more pronounced when the DoF exceeded 8.

To identify the cause of these delays, we measured the time taken for the `__x64_sys_read` function, which processes the read system call, to create `iomap`, `bio`, and `request` structures separately, as the DoF changes. As seen in Fig. 5, the `iomap` creation time slightly increased in proportion to the DoF, as one `iomap` is created per extent. The change in time spent on `bio` creation was smaller compared to that of `iomap` creation. This is because most of the `bio` creation time was spent allocating buffer pages, and the number of buffer pages to be allocated remains constant at 2048, regardless of the file's DoF. During the `request` creation process, if there are consecutive `bio` addresses, they will be merged into a single `request`. However, the fragmented file prevented it from being merged. Thus, the time spent on creating `requests` increased proportionally with the increase in the DoF.

Note that even under the extreme case where the DoF was 256, the kernel I/O path only took approximately 9.7 ms. In addition, when the I/O queue allows queueing of multiple outstanding commands, this I/O path delay can be mostly overlapped by the consecutive read operations to the following fragments. As shown in Fig. 4, if the `request` I/O queue depth was set to 128, which is the default value for a ramdisk, the file's DoF barely affected the time for the read operation.

We closely observed the kernel I/O path delay, which can be masked by I/O queueing. Fig. 6 shows the time spent on the `request` data structure creation and ramdisk access measured with `blktrace` [24] when reading a file with a DoF of 128 and a queue depth of 128. The kernel performs a *plug* process to merge `requests` for contiguous blocks, reducing the number of commands issued to the storage. The plugged `requests` are unplugged and sent to the device driver if the number of `requests` exceeds the predefined maximum pluggable `requests`, or if the size of an individual `request` surpasses the predefined plug flush size. The default values of these parameters are 16 `requests` and 128 KB, respectively.

Thus, during the experiment, 16 `requests` were plugged and then separately issued to the device driver since they all accessed separate blocks. As a result, the time spent creating the following 16 `requests` in the kernel I/O path is mostly masked by the time it takes the ramdisk to process the previous 16 `requests`. Additionally, by issuing multiple `requests` simultaneously, the processing time of the ramdisk is significantly reduced due to the operation overlap.

As a result of these experiments, we found that the delay occurring in the kernel I/O path due to request splitting is at the level of a few milliseconds, even in the extreme cases. Furthermore, we confirmed that its impact on actual execution time is negligible due to I/O operation queueing.

Next, we analyzed the execution time delay caused by request splitting in both the host-to-storage interface and the SSD inside. Since the implementation within the SSD is a black box, and it is impossible to accurately distinguish between the time consumed by the interface and the flash memory access time, we analyzed their combined execution time. For this analysis, we used two types of SATA SSDs and two types of NVMe SSDs, as shown in Table 1.

For this analysis, we performed a task to read 8 MB of contiguous data from the storage device by accessing the raw device file of the SSD to exclude the influence of the file system and kernel I/O path. In this process, we measured performance while increasing the unit read size from 32 KB to 8192 KB, doubling it each time. To minimize the impact of the SSD's state on the results, such as the ratio of invalid pages and the number of free blocks, we used the trim command for the entire area after each experiment to restore the SSD to the fresh-out-of-the-box (FOB) state. Then, we performed a sequential write on a 1 GB area to be read.

Fig. 7 shows the time taken to read 8 MB of data from each of the four SSDs, depending on the unit size of the read operation. Similar to Fig. 4, when the device's queue depth is set to 1, the elapsed time for reading 8 MB of data increases as the size of the read unit decreases. According to the regression analysis of the results, for NVMe SSDs, the elapsed time increased by 85 µs for each additional request, while for SATA SSDs, the time increased by 136 µs per request. These results encompass the impact of request splitting on the host-to-device interface, SSD firmware, and flash memory access time. Among these factors, the flash memory access time is expected to decrease as the unit read operation size increases influenced by the aforementioned internal parallelism.

However, the read time delay caused by request splitting

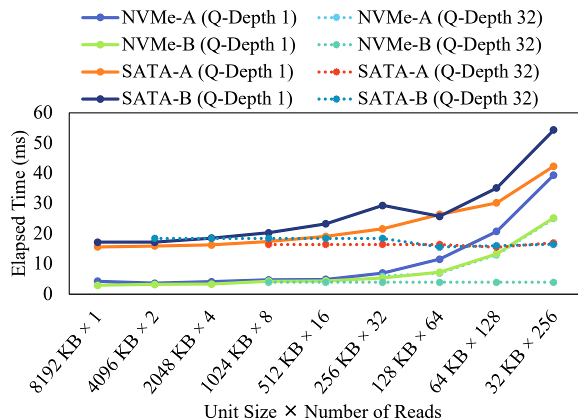


Figure 7: Time for reading 8 MB of data through raw device I/O operations with varying unit sizes.

disappeared when multiple outstanding I/O operations were queued, similar to the effect in the kernel I/O path. The native command queue (NCQ) of the SATA standard can queue up to 32 outstanding commands, while the NVMe standard also supports 65,536 queues with a queue depth of 65,535. Therefore, when request splitting occurs, the SSD can simultaneously place the split requests into the command queue and process them out of order. This reduces the number of interactions at the host-to-device interface and can further increase the die-level parallelism of the SSD. The dashed lines in Fig. 7 show the results when the command queue depth was set to 32. As expected, despite reducing the read unit size and, consequently, increasing the number of read commands in all SSDs, the difference in the total execution time was observed to be within a few milliseconds.

Based on our analyses, we confirmed that the request splitting overhead in the kernel I/O path is negligible compared to the increased operation time due to fragmentation. Furthermore, its impact is largely mitigated when issuing I/O operations asynchronously through command queueing. Additionally, we verified that even when request splitting occurs, the increase in processing time both at the host-to-device interface and within the storage device itself is extremely minimal, again thanks to command queueing, which most modern SSDs support.

3.2 Page Misalignment from Fragmentation

As explained with Fig. 2, when a file is written sequentially and there are no interrupting writes between the sequential write operations, the SSD evenly distributes the pages of the file across all dies in a round-robin manner. However, in cases of file fragmentation, such an ideal page allocation becomes impossible because writes to other files have occurred in between the writes to the fragmented file.

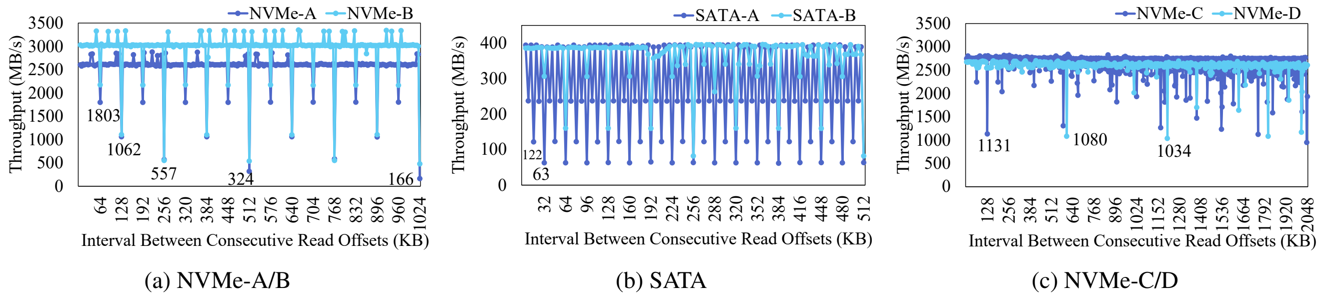
In cases of fragmentation, the page after the discontinuity point will be placed on a random die, regardless of the die where its semantically preceding page is located. In modern SSDs, because they have several tens of dies, the likelihood of a page containing a fragmented block being placed on a die immediately adjacent to the die where the previous file block's page is located is significantly low. As a result, when performing a sequential read access on a fragmented file, it causes significantly more die-level collisions compared to an ideal page placement scenario. The experiments in Fig. 3 created fragmented files and read them in such a way. Most previous research also fragmented files in the similar way. As a result, the significant performance degradation observed in fragmented file accesses in the experiments was very likely due to die-level collisions.

The read patterns observed at the die-level in these experiments can be emulated in actual SSDs by reading consecutively written file blocks at specific intervals. For instance, consider an SSD that assigns 4 KB pages to its dies in a round-robin manner and suppose this SSD has 16 dies. If we were to write 1 MB of data, that is, 256 pages consecutively, and then read every second page, resulting in reading 128 pages, this situation would produce a die-level read pattern similar to our experimental setup for sequentially accessing a 512 KB file with a DoF of 128. In this situation, compared to reading 128 consecutive pages without any interval, read operations would only take place on a half of the die set. This would inevitably lead to double the die-level collisions, making the time to read the 128 pages nearly twice as long. For the same reasons, reading every fourth page, amounting to 64 pages in total, would result in a read duration nearly four times longer than reading 64 consecutive pages.

To examine how read performance changes in such patterns, we conducted the following experiments. After initializing an SSD to its FOB state, we sequentially wrote 1 GB data to the area designated for reading. Subsequently, we configured fio to sequentially read 4 KB chunks at consistent intervals. For instance, if the interval of the read starting point were set to 16 KB, it would be set up to read 4 KB, skip a gap of 12 KB, and then read another 4 KB. To accomplish this, we modified the `blockalign` parameter of fio, incrementing it in 4 KB steps, ranging from a minimum of 4 KB to a maximum of 1024 KB in multiples. In these experiments, for NVMe SSDs, we set the *iodepth* parameter of fio to 512, and for SATA SSDs, we set it to the maximum supported value of 32. Furthermore, to exclude the effects of the file system and kernel I/O path, we configured fio to perform direct access on the raw device file.

Fig. 8a displays the throughput measurements for two NVMe SSDs when varying the read interval. When internal parallelism was adequately utilized, the SSDs achieved throughputs of 2600 MB/s and 3020 MB/s, respectively. However, as the interval between read operations expanded, the observed sustained throughput decreased to 166 MB/s and 480 MB/s for each SSD, respectively.

In both NVMe SSDs, the first significant performance drop was observed when the interval reached 64 KB. This indicates



(a) NVMe-A/B (b) SATA (c) NVMe-C/D

Figure 8: Throughput while varying the interval between starting points of consecutive read operations.

that the two NVMe SSDs allocate pages of a 32 KB size to a die before proceeding with allocation on the subsequent die, even though the actual number of pages allocated to a die at once might vary based on the device's page size. We refer to the size of the pages allocated to a single die at a given instance as the *die allocation granularity*. Both NVMe SSDs use a 16 KB page size [15,36], and a die allocation granularity of 32 KB means that they allocate two pages per die. This suggests that both SSDs store two pages per die using the two-plane program method [40].

The alignment size exhibited a significant performance drop starting at the 64 KB interval, and as it doubled each time, the decrease in performance became even more noticeable. This was likely due to the number of dies used for reads being halved every time the interval size doubled, as previously mentioned. In fact, for both SSDs, when the alignment size doubled from 64 KB to 1024 KB, the throughput decreased by 41 to 49% each time.

The lowest performance for both products was observed when the alignment size was 1024 KB for NVMe-A, dropping to approximately 6.5% of its typical value, and at 256 KB for NVMe-B, decreasing to 18.5%. These observations suggest that the *stripe size*, which represents the volume of data written across all dies before the allocation process restarts with the first die, differs among SSDs. Our experiments infer that the stripe size for NVMe-A is 1 MB, while for NVMe-B, it stands at 256 KB. For alignment sizes that exceed the stripe size, performance will mirror that of an alignment size equal to *(alignment size % stripe size)*.

This phenomenon was also observed in the SATA SSDs, as illustrated in Fig. 8b. While both products exhibited a throughput of 400 MB/s when all pages were accessible, the throughput decreased with the variation in the alignment size of accessible pages: dropping to 62 MB/s for SATA-A and 82 MB/s for SATA-B.

The performance degradation points of SATA SSDs showed a significant difference compared to those of NVMe SSDs, with SATA-A exhibiting its first performance drop at an alignment size of 8 KB. This indicates that its die allocation granularity is 4 KB. The most significant performance drop occurred at 32 KB, pointing to a stripe size of 32 KB. For SATA-B, the first performance drop was observed when the interval reached 32 KB, and, at 256 KB, it showed only 20.7% of its normal throughput. Therefore, SATA-B has a die allocation granularity of 16 KB, and the stripe size is estimated to be 256 KB.

When accounting for file fragmentation, the spacing between two accessed blocks typically aligns with multiples of the file block size, which is usually 4 or 8 KB. As evidenced in SATA SSDs, there's a marked performance dip when reading with intervals that are multiples of 4 KB. Consequently, the uptick in die-level collisions due to file fragmentation and the subsequent performance reduction are unavoidable. While NVMe SSDs generally have larger die allocation granularity and stripe sizes compared to SATA SSDs, leading to less pronounced performance drops with small read intervals, they are not exempt from the heightened die-level collisions brought about by the gap-reading patterns.

However, not all SSDs exhibited performance degradation at consistent intervals, as observed with the previous four products. Fig. 8c shows the performance drop in relation to the read offset intervals for NVMe-C and NVMe-D, respectively. Unlike the previous SSDs, the intervals at which these two products showed a decline were not necessarily powers of two. For NVMe-C, performance dips were noted at 64 KB and 128 KB intervals while the subsequent drops were found at multiples of 584 KB. In the case of NVMe-D, the drop was observed at intervals that are multiples of 604 KB. The die allocation policy of an SSD varies across manufacturers. However, the experimental results confirmed that non-sequential page access eventually leads to significant performance reduction due to high die-level collisions.

Unlike I/O path overhead or interface overhead that can be hidden by increasing the I/O queue depth, the read performance degradation due to die-level collisions was shown to persist even when the I/O queue depth was large. Therefore, for SATA SSDs with Linux kernel's default queue depth of 64 and NVMe SSDs with a queue depth of 1023, we can conclude that the main cause of performance loss due to file fragmentation is not the delay in the kernel I/O path or interface overhead but rather die-level collisions inside SSDs. In other words, while file fragmentation in HDDs causes additional seek time and rotational delay, in SSDs, it leads to additional die-level collisions.

4 Our Approach

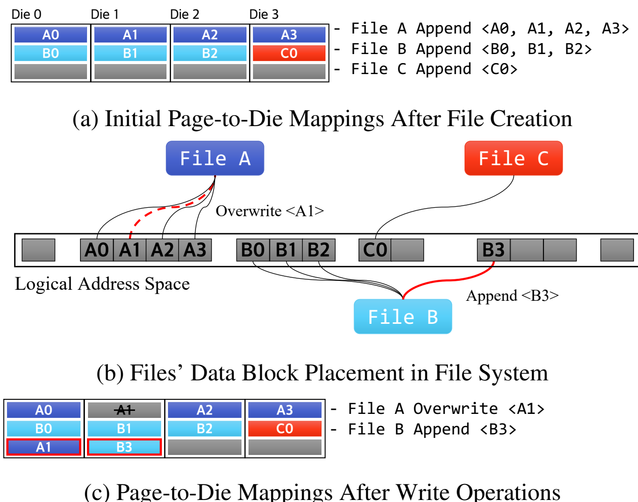
As previously analyzed, performance degradation of read operations to fragmented files mainly results from an increase in die-level collisions. However, the irregular page-to-die mapping observed in fragmented files is merely a consequence derived from the situations that cause fragmentation, rather than being a necessary condition for fragmentation to occur.

For example, let's assume that the three blocks of File B are written as in Fig. 9a and File C and A write a single block, respectively. From this initial state, if the application appends B3 to File B, B3 will be stored on the same die as B1, as shown in Fig. 9c. Consequently, a sequential read of File B will cause a die-level collision at Die 1. However, if File A overwrites not only A1 but also A2 and A3 in the situation shown in Fig. 9a, the position of B3 will shift by two dies and be located in Die 3. In this case, File B is stored in non-contiguous blocks on the file system, as shown in Fig. 9b, and sequential access to this file will cause request splitting. However, due to command queueing inside the SSD, all dies simultaneously process the same number of operations, enabling maximum performance. As previously mentioned, the time delay in the kernel I/O path and host-to-storage interface resulting from request splitting is minimal; consequently, despite File B being fragmented, its read performance remains barely affected.

Conversely, irregular page-to-die mapping may occur even without file fragmentation. Typically, overwriting an existing file block likely breaks the sequentiality of the page-to-die mapping. For instance, consider overwriting A1 in the situation shown in Fig. 9a. The file system supports in-place updates of blocks, so the position of A1 on the file system remains unchanged. Thus, File A maintains its contiguous state even after overwriting A1. However, since in-place updates of flash pages are impossible, the original page storing the A1 block becomes invalidated, and as shown in Fig. 9c, a new page for the updated A1 is assigned to Die 0, which follows Die 3. Consequently, although File A is contiguous at the file system level, a sequential read of File A will be significantly slowed down due to the die-level collision at Die 0.

Examining the two cases of fragmentation and overwriting that cause the irregular page-to-die mapping mentioned above, the fundamental reason is that SSD firmware cannot discern the file-level relationship between flash pages, and conversely, the file system cannot specify the position of the flash page storing the file block. To address this mismatch between page and file block placement, we propose an NVMe command extension and corresponding page-to-die mapping policy.

In our approach, the file system regards a write operation requiring a new data block allocation as an append write and the one to be performed on a data block already allocated to a file as an overwrite. If the Kernel I/O stack identifies the write being issued to the NVMe SSD as an append write or overwrite, it conveys additional information to the NVMe, on top of the existing NVMe write command, to perform



(a) Initial Page-to-Die Mappings After File Creation

(b) Files' Data Block Placement in File System

(c) Page-to-Die Mappings After Write Operations

Figure 9: File system-level block placement and storage-level page allocation of three files before and after write operations.

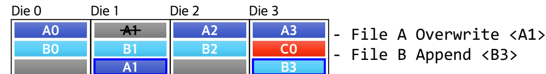


Figure 10: Page-to-die mappings after overwriting A1 and appending B3 blocks under our approach.

appropriate page-to-die mapping.

For append writes, the host provides the NVMe with the logical block address (LBA) of the file block immediately preceding the one being written, in addition to the write command. For example, when appending B3 to File B in Fig. 9a, the host sends the LBA of B2 along with the write command for B3 to the NVMe. In this case, the NVMe firmware deviates from the conventional round-robin algorithm for determining B3's die placement. Instead, as illustrated in Fig. 10, it assigns B3 to Die 3, which is the subsequent die after the one where B2 was stored. If the size of the write operation surpasses the die allocation granularity, the placement of additional pages adheres to the conventional round-robin approach; for instance, in this example, the second page is assigned to Die 0 after the first page is placed in Die 3.

For overwrites, the host sets a flag in the write command to indicate that the write operation is for overwriting an existing file block. For a write command with its overwrite flag set, the SSD firmware invalidates the existing flash page corresponding to the given LBA and allocates a new page. By assigning the new page to the same die where the original flash page was located, the die-level contiguity of the file blocks can be preserved. For example, when overwriting the A1 block of File A in Fig. 9a, a new flash page is allocated to Die 1, where the flash page storing A1 was originally located, as shown in Fig. 10, ensuring that sequential reads of File A maintain maximum internal parallelism. The die-level contiguity can also be preserved for overwrites exceeding the die allocation granularity by assigning new pages to the same dies where

the existing logical pages are located.

To implement the proposed approach, the host needs to provide additional information to the SSD when issuing a write operation. We can implement this without additional protocol overhead by utilizing unused bits in the NVMe protocol's write command. For example, the 24th and 25th bits of Command Dword (CDW) 12, which are currently unused and reserved, can be utilized to distinguish append writes and overwrites from conventional writes. Additionally, for append writes, the reserved CDW 2 and CDW 3 can be used to convey the LBA of the preceding file block.

Our approach specifically determines only the starting die for append writes, with subsequent writes following the existing mapping policy and being distributed across dies in a round-robin fashion. Consequently, it does not impact the performance of append writes. While it might be assumed that repetitive small-sized overwrites to the same file block could lead to write die collisions, these are typically merged in the host buffer and infrequently flushed to the SSD. Thus, even in these extremely rare cases, our approach does not adversely affect write performance.

Yet, continual overwrites on a small number of file blocks can quickly deplete free pages in certain dies, triggering GC earlier in these dies. Simultaneously, these overwrites invalidate the overwritten pages, reducing the number of valid pages. This decrease in valid pages necessitates fewer valid page copies during GC of those dies, which not only lowers the write amplification factor but also shortens the duration of the GC process.

However, despite these conditions being rare, they are not ideal, as they can lead to more frequent GCs in specific dies and cause uneven wear across the dies. This uneven wear might result in some dies wearing out prematurely, ultimately shortening the lifespan of the SSD. The LBAs that are targets of the frequent overwrites are likely to be evenly distributed and allocated across multiple dies, minimizing the occurrence of uneven wear. Nonetheless, should wear disparity become significant, a mechanism to reallocate the page for that specific LBA to a different die for wear leveling would mitigate the situation, albeit potentially at the expense of performance.

5 Evaluation

To assess the validity and efficacy of the proposed approach, we carried out two evaluations. First, to validate the proposed scheme, we emulated the write patterns as if the proposed approach were applied in commodity SSDs, and measured the read performance. Second, to examine the performance benefits that applications can gain through the proposed scheme, we implemented it in the ext4 file system and the NVMeVirt SSD emulator.

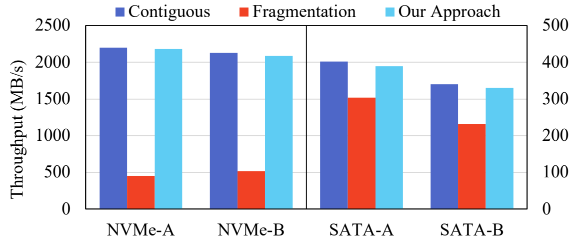
5.1 Validation of Our Approach

To implement the proposed scheme, the SSD's NVMe protocol stack must be modified to process the NVMe command extension, and its page-to-die mapping policy should also be adjusted to utilize the hints provided by the host through the command extension. However, modifying actual SSD firmware is not feasible. Therefore, to verify the validity of our approach, we created write patterns that would result in the same page-to-die mapping as the proposed approach under file fragmentation and partial file overwrite situations. We then measured the read performance of the files written in this manner. For these experiments, we deferred file system metadata writes to prevent them from interfering with die allocation control and configured journaling to be performed on a separate storage device. In these experiments, we used NVMe-A, NVMe-B, SATA-A, and SATA-B, all of which have regular die allocation granularity and stripe size, as depicted in Fig. 8.

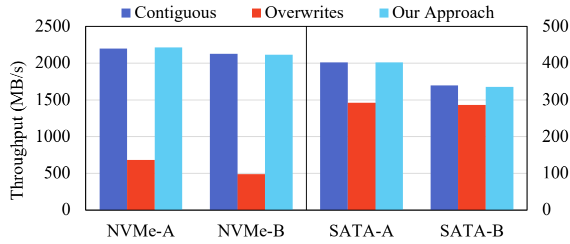
To evaluate the effectiveness of our proposed method under fragmentation, we generated a fragmented file on an FOB-state SSD and measured its read time. This file was formed by appending 256 fragments, each sized according to the SSD's die allocation granularity, cumulating to an 8 MB file. After writing each fragment of the die granularity size, if we write enough data to the dummy file to fill the remaining space of the SSD's stripe before writing the next fragment, all the fragments of the target file will be assigned to one die, resulting in significant performance degradation, which is denoted as *Fragmentation* in Fig. 11. Due to the smaller die allocation granularities of the SATA SSDs, 256 appends were insufficient to reach the desired 8 MB file size. To compensate, we adjusted the final append's size to ensure the total file size was 8 MB. Note that, as a result, only the initial segment of the file became fragmented in the SATA SSDs.

In order to emulate the proposed approach, we first wrote a single fragment and then wrote an amount of garbage data equal to the SSD's stripe size to a dummy file. Subsequently, we wrote the next fragment to the target file. Repeating this process, as shown in Fig. 10, each fragment of the target file would be located in the die immediately following the die where the previous fragment was located. Thus, while fragmentation occurs at the file system level, within the SSD, flash pages would be sequentially assigned to consecutive dies. We repeatedly read the written fragmented file while measuring the throughput.

As illustrated in Fig. 11a, the read performance of fragmented files on NVMe SSDs degraded by 79% for NVMe-A and 76% for NVMe-B, in comparison to that of contiguous files. Since the file was appended 256 times in 32 KB sizes on the NVMe SSDs, it was stored entirely on a single die. This led to die-level collisions during most read operations. We believe that NVMe-A's larger performance decrease was attributed to it having more dies than NVMe-B. Our approach



(a) Append Write



(b) Overwrite

Figure 11: Read performance of four kinds of SSDs for contiguous files, fragmented files, and fragmented files under our proposed approach, respectively.

mitigated this, with performance experiencing only a 2% decline from that of the contiguous files.

In the case of the SATA SSDs, the performance degradation due to fragmentation was less severe than that of NVMe because only the frontal part of the file was fragmented due to their smaller die allocation granularity size. Although the portions of the target files that were fragmented amounted to 12.5% for SATA-A and 50% for SATA-B, the performance experienced a degradation of 27% and 16%, respectively, when compared to the contiguous cases. In both products, our approach reduced the performance degradation, achieving nearly the same performance as accessing contiguous files, with only a 1.2% difference.

To understand the misalignment in page-to-die mapping caused by overwrites on a file, and the resultant performance degradation from die-level collisions, we conducted experiments on the four types of SSDs. First, we created a file. Then, we performed 256 overwrites, each of 32 KB, from the beginning to the end of the file. Finally, we read the file. The size of the target file was again set to 8 MB. Between consecutive overwrite operations, we wrote random data as large as (stripe size - die allocation granularity).

After finishing the series of overwrite operations, the file's pages would be placed in a single die, which is denoted as *Overwrites* on the graph in Fig. 11. Note that the file's data blocks will remain contiguous at the file system level even after the overwrites are performed.

Fig. 11b shows the results for the overwrite experiments. The performance degradation of the NVMe SSD was similar to that of a fragmented file, showing a significant performance drop to a quarter. However, when our approach was applied, the performance degradation was reduced to an average of 1% compared to the contiguous case. In the case of SATA SSDs, their performance degradation was smaller due to the difference in die allocation granularity mentioned earlier, but they still showed 27% and 16% decrease in performance, respectively. However, our approach was able to successfully achieve a similar level of performance as before the overwrites were executed. The efficacy of our approach was observed for all four SSDs. The largest performance degradation under our approach was merely 1.2% for SATA-B.

From this analysis, we confirmed that the proposed approach can effectively prevent the loss of read performance even for heavily fragmented files and also successfully avoid read performance degradation caused by overwrites.

Table 2: Parameters used for NVMe emulation.

SSD	Capacity	60 GB
	Host Interface	PCIe Gen3 $\times 4$
	FTL L2P Mapping	Page Mapping [1, 6]
	Channel Count	4
	Dies per Channel	2
Flash Memory [22]	Read/Write Unit Size	32 KB
	Read Time	36 µs
	Write Time	185 µs
	Channel Speed	800 Mbps

5.2 Effectiveness for Application Workloads

To evaluate the holistic effectiveness of the proposed approach, we implemented the host-side part of the proposed scheme both in the ext4 file system and the Linux kernel's NVMe device driver.[1] This allowed applications to directly utilize our approach via the file system. On the SSD side, we implemented our proposed approach's write command extension and page-to-die allocation mechanism within NVMeVirt.[2] The parameters for NVMeVirt were sourced from Table 2. The die allocation granularity for the emulated SSD was set at 32 KB, and the stripe was set to 256 KB, which mirrors the settings of NVMe-B. To mitigate the onset of fragmentation, the ext4 file system was adjusted in accordance with the experimental configurations delineated in Section 3.

First, we executed experiments based on the configuration depicted in Fig. 11 using the aforementioned implementation. In contrast to prior experiments using actual SSDs that required meticulous control over dummy write sizes, the implementation can sustain optimal die mapping even when random offsets interleave between successive file block writings. This enabled us to extend our analysis beyond just the worst-case conditions, incorporating cases more reflective of

[1] The source code of the NVMe driver and ext4 extension implemented in the Linux kernel can be accessed at https://github.com/yuhun-Jun/kernel_5_15_DA.

[2] The SSD emulator enabled with our approach can be found at https://github.com/yuhun-Jun/nvmevirt_DA.

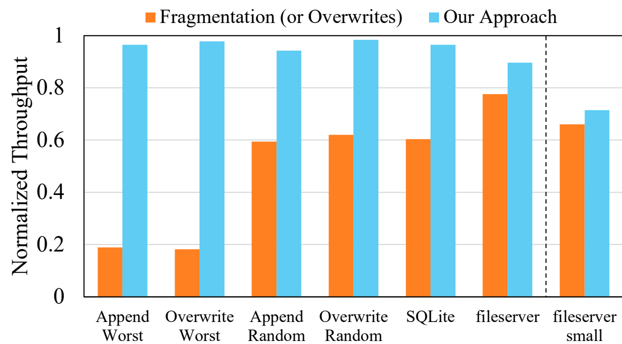


Figure 12: Normalized read throughput of applications executed with the implementation of our approach relative to that with ideal file block and flash page placement.

real-world operations where intervening dummy writes between target file writes are of random sizes. Consequently, instead of confining file blocks' pages to a single die as in earlier experiments, they were distributed randomly across all dies after finishing the append or overwrite operations.

In the worst-case experiments, where all file blocks were allocated to a single die due to fragmentation or overwriting, the results shown in Fig. 12 aligned with those seen in Fig. 11. We observed a significant drop in read performance, which stood at only 19% of the normal throughput for fragmented files and 18.2% for overwritten ones. Our proposed approach successfully preserved the read performance after the append and overwrite operations and only resulted in a 3.5% performance dip for *Append Worst* and 2.3% for *Overwrite Worst*.

In the random disturbance experiments, where the size of dummy file writes between target file writes varied randomly at the 32 KB granularity, ranging from 32 KB to 32 MB, the performance decreased to 59.4% of the ideal for the fragmenting append experiment and 62% for the overwrite experiment. Our approach again successfully suppressed the read performance degradation to 5.8% for *Append Random* and 1.6% for *Overwrite Random*.

In addition to the hypothetical workloads, we analyzed the effectiveness of the proposed approach with SQLite [29] and Filebench's fileserver workload [38].

We established a table and inserted 10,000 records, each 16 KB in size, with SQLite. Simultaneously, we appended 100 KB chunks repeatedly to a dummy file. Following this, we executed a `select` query to retrieve all 10,000 records from the resulting database file, which had a DoF of 5,005. As depicted in the *SQLite* column of Fig. 12, the select query's performance was only 60% of the case where no disturbing writes were performed. In contrast, under our approach, the database file blocks were stored on consecutive dies as intended even with the existence of the dummy writes. As a result, we observed a performance increase of 1.6 times, which represents only a 3.5% drop compared to the case without fragmentation.

The fileserver workload mimics the I/O patterns of a file server. For this, it employs multiple threads executing file creation, random-sized append writes of up to 16 KB, sequential reads on random files, and random file deletion on a file set consisting of 10,000 files averaging 128 KB each. To induce more severe file fragmentation, we modified the workload so that it preallocates a file set of 10,000 128 KB files, each of 10 threads performs 32 KB size append writes on random files from the file set for a duration of 1 minute and measured the read performance. We also removed the file creation and deletion from the workload. At the end of the experiment, the average file size was around 600 KB, the average DoF was 15.7, and the total file set size was 11 GB. The results showed a read performance at 80% of the level seen when files were stored in contiguous file blocks. This lesser performance degradation compared to the previous experiments was due to multiple threads reading simultaneously, increasing the number of outstanding commands. This ensured that most dies continually received operations, enhancing die-level parallelism. Our approach was able to recover the sequential read performance on fragmented files to 93% of the ideal file placement condition.

The *fileserver small* shown in Fig. 12 is from an experiment with settings identical to *fileserver*, but where the append write size was set to 16 KB, smaller than the die granularity. In this experiment, fragmentation further reduced read performance. When writing 32 KB chunks, a single flash page, of which size is 32 KB, can accommodate one write request. However, when writing 16 KB chunks, two chunks are combined and written to a single flash page. As a result, writes from two different files could be recorded on the same page, meaning files of the same size ended up being stored across more pages. This leads to a higher number of flash page reads when reading the file. We confirmed that this phenomenon also occurs when a file uses `fallocate` to pre-allocate consecutive file system blocks and then fills in data in small increments to make a contiguous file, especially if small writes for dummy files intervene. This serves as further evidence that the fragmentation-induced performance degradation is not directly due to fragmentation but rather an issue of data placement within the SSD.

This experiment underscores the limitations of the proposed approach. While it is designed to achieve consecutive die allocation of file blocks during file fragmentation, it's not equipped to counteract the effects of small intervening writes, leading to a file write potentially spanning multiple pages. Consequently, its performance enhancement stood at 8.2%. Addressing the flash page-level fragmentation issue, which may also occur to contiguous files at file system level when the write size is smaller than the flash page size, requires a novel page allocation strategy to counteract that. Such a study would go beyond the scope of this paper and points to an interesting topic for future research.

6 Conclusion

Contrary to early beliefs that file fragmentation does not impact SSD performance, it has now been recognized that SSDs can indeed suffer significant declines in read performance due to file fragmentation. In this paper, we have shown that the root cause of this performance degradation is not delays in the kernel I/O path caused by request splitting, as previously described in the literature. Instead, it arises from misalignments in the SSD's page-to-die mapping, which increase die-level collisions. Furthermore, we demonstrated that such misalignments can occur not only during file fragmentation but also when files are overwritten.

To address this issue, we proposed an NVMe command extension that enables the file system to provide hints about the write operation to SSDs, as well as a novel page-to-die mapping scheme considering the hints for the SSD controller. This ensures that pages are allocated to contiguous dies based on their order in the file. The resulting well-ordered page-to-die mappings effectively prevent additional die-level collisions caused by both file fragmentation and overwrites. Our evaluation showed that, without resorting to costly defragmentation or file rewriting, the proposed approach effectively suppresses the read performance degradation for fragmented or overwritten files to a mere few percent.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Peter Desnoyers, for their valuable suggestions for this paper.

This research was supported by Samsung Electronics, and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (2021R1A2C200497612).

References

[1] Amir Ban. Flash file system, April 4 1995. US Patent 5,404,485.

[2] Mingming Cao, Theodore Y Tso, Badari Pulavarty, Suparna Bhattacharya, Andreas Dilger, and Alex Tomas. State of the art: Where we are with the Ext3 filesystem. In *Proceedings of the Ottawa Linux Symposium (OLS 05)*, pages 69–96. Citeseer, 2005.

[3] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture (HPCA 11)*, pages 266–277. IEEE, 2011.

[4] 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 *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 45–58, 2017.

[5] 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 *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[6] Intel Corporation. Understanding the flash translation layer (FTL) specification, 1998.

[7] Giel de Nijs, Ard Biesheuvel, Ad Denissen, and Niek Lambert. The effects of filesystem fragmentation. In *Proceedings of the Linux Symposium*, volume 1. Citeseer, 2006.

[8] BTRFS documentation. btrfs-filesystem(8). https://btrfs.readthedocs.io/en/latest/btrfs-filesystem.html.

[9] NVM Express. NVMe Base Specification Revision 1.4c. 2021.

[10] Windows 8 Help Forums. Optimize drives - defrag HDD and TRIM SSD in Windows 8. https://www.eightforums.com/threads/optimize-drives-defrag-hdd-and-trim-ssd-in-windows-8.8615/.

[11] Congming Gao, Liang Shi, Mengying Zhao, Chun Jason Xue, Kaijie Wu, and Edwin H.-M. Sha. Exploiting parallelism in I/O scheduling for access conflict minimization in flash-based solid state drives. In *Proceedings of the Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2014.

[12] Samsung Semiconductor Global. SSD performance FAQs | support. https://semiconductor.samsung.com/consumer-storage/support/faqs/03/.

[13] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, Jihong Kim, et al. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *Proceedings of the USENIX Annual Technical Conference(ATC 17)*, pages 759–771, 2017.

[14] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the international conference on Supercomputing (ICS 11)*, pages 96–107, 2011.

[15] Woopyo Jeong, Jae-woo Im, Doo-Hyun Kim, Sang-Wan Nam, Dong-Kyo Shim, Myung-Hoon Choi, Hyun-Jun Yoon, Dae-Han Kim, You-Se Kim, Hyun-Wook Park, et al. A 128 Gb 3b/cell V-NAND flash memory with 1 Gb/s I/O rate. *IEEE Journal of Solid-State Circuits*, 51(1):204–212, 2015.

[16] Cheng Ji, Li-Pin Chang, Sangwook Shane Hahn, Sungjin Lee, Riwei Pan, Liang Shi, Jihong Kim, and Chun Jason Xue. File fragmentation in mobile devices: Measurement, evaluation, and treatment. *IEEE Transactions on Mobile Computing*, 18(9):2062–2076, 2018.

[17] Cheng Ji, Li-Pin Chang, Liang Shi, Chao Wu, Qiao Li, and Chun Jason Xue. An empirical study of file-system fragmentation in mobile storage systems. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[18] Yuhun Jun, Jaehyung Park, Jeong-Uk Kang, and Euiseong Seo. Analysis and mitigation of patterned read collisions in flash SSDs. *IEEE Access*, 10:96997–97009, 2022.

[19] 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, 2010.

[20] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R Ganger. Geriatrix: Aging what you see and what you don't see. a file system aging approach for modern storage systems. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC 18)*, pages 691–704, 2018.

[21] Jeong-Uk Kang, Jin-Soo Kim, Chanik Park, Hyoungjun Park, and Joonwon Lee. A multi-channel architecture for high-performance NAND flash-based storage system. *Journal of Systems Architecture*, 53(9):644–658, 2007.

[22] Sang-Hoon Kim, Jaehoon Shim, Euidong Lee, Seongyeop Jeong, Ilkueon Kang, and Jin-Soo Kim. NVMeVirt: A versatile software-defined virtual NVMe device. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST 23)*, Santa Clara, CA, February 2023.

[23] Aneesh Kumar KV, Mingming Cao, Jose R Santos, and Andreas Dilger. Ext4 block and inode allocator improvements. In *Proceedings of the Linux Symposium*, volume 1, 2008.

[24] Linux man page. blktrace(8). https://linux.die.net/man/8/blktrace.

[25] Linux man page. e4defrag(8). https://man7.org/linux/man-pages/man8/e4defrag.8.html.

[26] Linux man page. filefrag(8). https://man7.org/linux/man-pages/man8/filefrag.8.html.

[27] Linux man page. xfs_fsr(8): filesystem reorganizer for XFS. https://man7.org/linux/man-pages/man8/xfs_fsr.8.html.

[28] 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. Citeseer, 2007.

[29] SQLite Home Page. Sqlite. https://sqlite.org/index.html.

[30] Debian Man pages. defrag.f2fs(8) — f2fs-tools — debian testing. https://manpages.debian.org/testing/f2fs-tools/defrag.f2fs.8.en.html.

[31] Jonggyu Park and Young Ik Eom. Fragpicker: A new defragmentation tool for modern storage devices. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 21)*, pages 280–294, 2021.

[32] Jonggyu Park and Young Ik Eom. File fragmentation from the perspective of I/O control. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage 22)*, pages 126–132, 2022.

[33] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The linux B-tree filesystem. *ACM Transactions on Storage*, 9(3):1–32, 2013.

[34] SATA-IO. Serial ATA Revision 3.0, 2009.

[35] Takashi Sato. Ext4 online defragmentation. In *Proceedings of the Linux Symposium*, volume 2, pages 179–86. Citeseer, 2007.

[36] Chang Siau, Kwang-Ho Kim, Seungpil Lee, Katsuaki Isobe, Noboru Shibata, Kapil Verma, Takuya Ariki, Jason Li, Jong Yuh, Anirudh Amarnath, Qui Nguyen, Ohwon Kwon, Stanley Jeong, Heguang Li, Hua-Ling Hsu, Tai-yuan Tseng, Steve Choi, Siddhesh Darne, Pradeep Anantula, Alex Yap, Hardwell Chibvongodze, Hitoshi Miwa, Minoru Yamashita, Mitsuyuki Watanabe, Koichiro Hayashi, Yosuke Kato, Toru Miwa, Jang Yong Kang, Masatoshi Okumura, Naoki Ookuma, Muralikrishna Balaga, Venky Ramachandra, Aki Matsuda, Swaroop Kulkani, Raghavendra Rachineni, Pai K. Manjunath, Masahito Takehara, Anil Pai, Srinivas Rajendra, Toshiki Hisada, Ryo Fukuda, Naoya Tokiwa, Kazuaki Kawaguchi, Masashi Yamaoka, Hiromitsu

Komai, Takatoshi Minamoto, Masaki Unno, Susumu Ozawa, Hiroshi Nakamura, Tomoo Hishida, Yasuyuki Kajitani, and Lei Lin. A 512Gb 3-bit/cell 3D flash memory on 128-wordline-layer with 132MB/s write performance featuring circuit-under-array technology. In *Proceedings of the International Solid-State Circuits Conference (ISSCC 19)*, pages 218–220. IEEE, 2019.

[37] Keith A Smith and Margo I Seltzer. File system aging—increasing the relevance of file system benchmarks. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS 97)*, pages 203–213, 1997.

[38] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[39] Micron Technology. Should you defrag an SSD? https://www.crucial.com/articles/about-ssd/should-you-defrag-an-ssd.

[40] Micron Technology. Tn-29-28: Memory management in NAND flash arrays overview. 2005.

[41] Tenforums. Optimize and defrag drives in Windows 10. https://www.tenforums.com/tutorials/8933-optimize-defrag-drives-windows-10-a.html.

[42] Guangyu Zhu, Jeongeun Lee, and Yongseok Son. An efficient and parallel file defragmentation scheme for flash-based SSDs. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (SAC 22)*, pages 1208–1211, 2022.

Artifact Appendix

Abstract

The provided artifacts consist of shell scripts designed to replicate the experimental results introduced in the paper. These experiments were aimed at analyzing read performance degradation caused by fragmentation and assessing the efficacy of the proposed approach. Additionally, the artifacts include a customized NVMeVirt implementation featuring the proposed page placement scheme for the FTL. This is complemented by a modified Linux kernel equipped with the necessary file system and NVMe device driver support for the customized NVMeVirt.

Scope

These artifacts include shell scripts that enable the replication of results presented in Section 3, as depicted in Figs. 3, 4, 7, and 8. The shell scripts for demonstrating the effectiveness of the proposed approach, as illustrated in Fig. 11, are also included in the artifacts.

Furthermore, the artifacts comprise a customized NVMeVirt utilizing an FTL that implements the proposed page placement scheme. This is complemented by the modified Linux kernel, which is also a part of the artifacts. It provides NVMeVirt with the page placement hints. The shell scripts for conducting experiments on the workloads used in Section 5.2 are provided as well. These scripts were instrumental in obtaining the experimental results showcased in Fig. 12.

Contents

The shell scripts below run the experiments introduced in Section 3 and Section 5.1. In the file names, the `*` is replaced with the target device name, such as `NVMe_A` or `SATA_B`.

`varyingdof_*.sh`: These shell scripts are for the experimentation analyzing the read time change according to the varying DoF of files stored on NVMe and ramdisk, as shown in Figs. 3 and Fig. 4, respectively.

`interface_*.sh`: These measure the time taken to read 8 MB of data from the target SSD, depending on the unit size of the read operation. This was used to produce the results illustrated in Fig. 7.

`alignment_*.sh`: These measure the throughput of read operations while varying the interval between starting points of consecutive operations, as shown in Fig. 8.

`pseudo_(append|overwrite)_*.sh`: These shell scripts mimic the write patterns for three cases: when files are written contiguously, when written in a fragmented manner, and when written according to the write patterns that occur in our approach. It then measures the read performance for each of these cases. This was used to produce the results introduced in Fig. 11.

The following shell scripts are intended for the experiments explained in Section 5.2.

`hypothetical_(append|overwrite).sh`: These shell scripts measure the read throughput for a file after performing a series of append write or overwrite operations to it. The append and overwrite operations can be configured to follow the worst-case pattern or the random pattern. The results of executing these on NVMeVirt are shown in Fig. 12.

`sqlite.sh`: This was used to obtain the experimental results shown in Fig. 12. It triggers write operations to create a fragmented database file when running SQLite. Subsequently, it performs select operations through SQLite on the fragmented database file and measures the performance.

`fileserver.sh`: This was also used to obtain the experimental results shown in Fig. 12. This shell script measures the performance in circumstances where the files generated by Filebench's fileserver workload become fragmented.

`fileserver_small.sh`: This script is similar to `fileserver.sh`, except that the append operations are performed with a size smaller than the flash memory page size.

The detailed instructions can be found in the `README.md` file located in the GitHub repository.

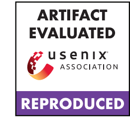
Hosting

The GitHub repository for the artifacts is https://github.com/yuhun-Jun/fast24_ae. The results introduced in this paper were produced from the commit version `89ba3a9` of the `main` branch.

Requirements

The shell scripts for analyzing fragmentation-induced performance degradation must be configured according to the internal parameters of the target SSD. The provided artifacts are set up for the devices introduced in Table 2. For other SSDs, settings including the write offset must be appropriately adjusted.

For the customized NVMeVirt to function properly, the support from OS Kernel's file system and NVMe driver is mandatory. Therefore, it operates correctly only when executed on the provided Linux Kernel. Furthermore, as NVMeVirt utilizes main memory to emulate storage space, stable experimental outcomes require that the workload operates exclusively within a single NUMA domain. This approach avoids cross-NUMA domain memory accesses, which can significantly vary in execution time and potentially affect the consistency of results.



In-Memory Key-Value Store Live Migration with NetMigrate

Zeying Zhu⋆*, Yibo Zhao*†*, Zaoxing Liu*⋆

⋆*University of Maryland,* †*Boston University*

Abstract

Distributed key-value stores today require frequent key-value shard migration between nodes to react to dynamic workload changes for load balancing, data locality, and service elasticity. In this paper, we propose NetMigrate, a live migration approach for in-memory key-value stores based on programmable network data planes. NetMigrate migrates shards between nodes with zero service interruption and minimal performance impact. During migration, the switch data plane monitors the migration process in a fine-grained manner and directs client queries to the right server in real time, eliminating the overhead of pulling data between nodes. We implement a NetMigrate prototype on a testbed consisting of a programmable switch and several commodity servers running Redis, and evaluate it under YCSB workloads. Our experiments demonstrate that NetMigrate improves the query throughput from 6.5% to 416% and maintains low access latency during migration, compared to the state-of-the-art migration approaches.

1 Introduction

Modern internet services (e.g., e-commerce, mobile gaming, and social networks) depend on large-scale key-value stores as the backend to perform various jobs (e.g., web caching, real-time analytics, and machine learning) [2, 15, 24, 27, 38, 49]. These services often require databases to process queries over ever-growing data volumes and dynamic workload distributions. However, static sharding limits the ability of such systems to adapt to rapidly changing workloads. This may result in degraded performance and Service Level Agreement (SLA) violations due to load imbalance, poor data locality, and insufficient provisioning of cloud resources [28, 33, 37, 51, 52]. Live migration techniques are widely adopted for key-value store reconfiguration [28, 33, 37, 41, 53] that migrates data between nodes without service downtime.

Existing live migration techniques must assume one or more locations — migration source, destination, or both, as the main query serving point because the actual location of a key-value pair during migration is unknown. *Source-based solutions* [23, 33, 53] use source storage to serve all client queries during migration and incrementally migrate dirty data logs to the destination when the keys are updated at the source after their migration. Alternatively, *destination-based solutions* [28, 37] transfer data ownership at the beginning of migration and immediately routes newly arrived queries to the destination. The not-yet-migrated data queried by the client are pulled on demand from the source. However, while two techniques can continuously serve user queries during migration and aim to achieve minimal downtime, they suffer from fundamental performance limitations. Lack of insight about the migration process is a key roadblock in minimizing the overhead caused by accessing the data at the location that does not have ownership. For example, it takes another round-trip latency to fetch a key-value pair from the source if destination-based migration is adopted and this pair has yet migrated to destination.

Ideally, if client queries can always be served at the "right" location during migration, the cost to serve the queries would be minimized. Considering either source- or destination-based migration, extra data movements between the source and destination are necessary when the queried data are not present at the original location. To address this problem, *hybrid migration* techniques take advantage of both source and destination to serve user queries by tracking the migration process in clients [30] or replicate queries to both source and destination [41]. While leveraging both source and destination for query serving during migration is promising for achieving better performance, the cost at the client side to track data ownership and the potential overhead between clients and servers for maintaining consistency are nontrivial.

In this paper, we propose **NetMigrate** to rethink the problem of designing a hybrid live migration approach for in-memory key-value stores. NetMigrate aims at leveraging either source or destination who owns the accessed data chunks (migration state) to achieve pauseless migration and minimal impact on query performance. Compared to bookkeeping of detailed migration states on the client or with additional resources, we argue that the network itself (e.g., top-of-rack switches) would be a better place to track the migration process on the fly because they have a central view of all the data movements in the dedicated rack-scale clusters. With emerging programmable hardware, ToR switches can be programmed to track migration states (e.g., which key-value pairs are migrated) at line rates without latency overhead and can directly route the client queries to the right location (source or destination) who holds the up-to-date data. To our knowledge, NetMigrate is the first proposal to leverage in-network programmability to improve storage migration.

Realizing the promise of NetMigrate, however, is easier said than done, and the use of programmable switches for

migration has several key challenges:

- ***Tracking migration states with limited on-switch resources.*** While existing programmable switches guarantee high-line-rate packet processing capability (e.g., Tbps), the on-switch resources for performing packet-level operations are limited, e.g., O(10MB) SRAM and limited accesses to the SRAM [7, 45]. Given the resource constraints, we leverage probabilistic data structures, i.e., Bloom filters [25] to track if a key has finished migration and counting Bloom filters [29] to track if a key is currently under migration, as the "indexing service" to record the up-to-date migration status. Moreover, to migrate a storage instance with a large number of keys, we support dynamically adjusting the monitoring granularity from a per-key level to a level of a group with multiple keys. With these techniques, 64 MB SRAM on switch will be able to support up to a 34-billion-key storage cluster migration in the same rack[1](§4.2).
- ***Maintaining data consistency during migration.*** To ensure strong consistency, it is critical to understand the location that holds the most updated value of a key and route the new queries to it during migration. It is challenging to keep this information because of the pending state of ongoing migration between source and destination and additional errors from probabilistic migration state tracking. We design an error handling method to ensure correct query results, following this principle: If we have absolute confidence about data ownership, the switch routes the queries to the corresponding location; otherwise, the switch issues small numbers of replicated queries (e.g., double reads) when there is any possibility of imprecise information (§4.3).
- ***Supporting diverse migration policies.*** Some features of existing migration protocols can be useful for certain migration scenarios. For instance, operators may prefer to use destination-based Rocksteady [37] to ensure a short migration time because the resource on the source server can be used primarily for migrating data. NetMigrate can tune its on-switch data structures and resource budgets from the source side to optimize various performance goals, such as minimizing migration time and maximizing query throughput. With the help of switches, NetMigrate can be adjusted to achieve comparable migration time as destination-based solutions such as Rocksteady while offering better query throughput and latency (§4.4).

We implement a NetMigrate prototype in the P4 language [10] (switch side) and C++ (client and server side), and evaluate it on a testbed with an Intel Tofino switch and 3 commodity servers. We migrate a Redis key-value store [11] as an example consisting of 256 million key-value pairs with 4-Byte keys and 64-Byte values, and evaluate NetMigrate on YCSB workloads [17] with different write ratios and load-balancing scenarios against the state-of-the-art approaches. Experimental results demonstrate that NetMigrate achieves zero downtime during migration, improves the average query throughput by 6.5% to 416%, while maintaining low access latency during migration and incurring negligible extra bandwidth overhead. NetMigrate is open-sourced at [9].

[1] As in § 4.2, assuming 16 bits per element in Bloom filter and counting Bloom filter and each group has 2^{10} keys, 64MB SRAM can support 34 billion key-value pairs (64MB*8/16*1024). With 4-Byte keys and 64-Byte values, the total storage size is ∼2TB.

2 Background and Motivation

In this section, we first discuss the key-value store live migration problem and its requirements. We then analyze existing approaches and their advantages and limitations.

2.1 Key-Value Store Live Migration

In distributed key-value store systems, data sharding can be reconfigured over time for load balancing, access-locality improvement, and cluster horizontal scaling (e.g., when a new node joins the cluster, it "steals" keys from other nodes). Storage instance migration improves spatial locality to enhance item access throughput and reduce access latency to backend servers [20, 28, 34, 37] and provide load balancing among dynamic and skewed workloads between servers [28, 32, 42].Migration can also happen when there is an upgrade of the server hardware or cluster horizontal scaling.

Data migration between shards can introduce service downtime and performance degradation. However, during migration, storage cluster should still provide service reliability and meet the Service Level Agreement (SLA). For example, even a slight service outage has significant financial consequences for a large-scale e-commerce platform and can harm the customer's trust [27]. Thus, live migration techniques, which move data between nodes without causing client-observable downtime, become a key enabler to achieve elastic key-value stores in the cloud environment.

In this paper, we focus on live migration of in-memory key-value stores, such as Redis [11], Apache Cassendra [1], RAMCloud [46], and Memcached [8]. These key-value stores keep all data in DRAM and can scale across thousands of data center servers. Under these storage systems, they often construct hash-table data structure for storing and indexing key-value pairs. We focus on alleviating migration performance degradation to the minimum. We assume that there is an internal or external cluster scheduler that collects statistics of the storage cluster and generates reconfiguration plan on when and how the data should be re-sharded and migrated to fit the current workloads.

Performance requirements. Common metrics used to evaluate a live migration system include service downtime, query throughput and latency, transferring extra data (extra network bandwidth usage), and migration completion time. For migration approaches, there is a fundamental trade-off between migration completion time and migration cost (e.g., drop in query performance and transfer of extra data). The shorter the migration finish time, the higher the migration cost. An

Migration Protocols	Example Systems	Downtime	Latency	Throughput	Extra Data	Client Overhead	Migration Time
Stop-and-Copy	Redis MIGRATE [13], Slacker [23]	Yes	High	Low	No	No	Short
Source-based	RAMCloud [46], Remus [33], DrTM+B [53]	Minimal	Low	Medium	Yes	No	Long
Destination-based	Rocksteady [37]	No	High	Low	No	Yes	Short
Hybrid	Fulva [30]	No	Medium	Medium	No	Yes	Medium
Hybrid	**NetMigrate**	**No**	**Low**	**High**	**Negligible**	**Negligible**	**Medium (Adjustable)**

Table 1: **Overview of live migration approaches.**

ideal live migration approach is expected to provide minimal migration cost while maintaining an acceptable migration finish time.

2.2 Existing Approaches and Limitations

Stop-and-copy is a basic form of migration, which consists of freezing the storage server (with a read lock), copying the key-value data to the destination server, and then deleting them in the source server. For example, Redis MIGRATE command [13] implements this stop-and-copy at the per-key level. If migrating the entire store to the destination server, a faster way is to shutdown the source key-value store, create a snapshot file, perform a file-level copy of the compressed snapshot to destination, and then start a new key-value store instance on the destination server pointing to the copied snapshot directory. The main downside of stop-and-copy is the significant downtime caused by shutting down the storage instance, which affects the client execution logic. The length of copying period is proportional to the database size [23].

To perform live migration, there are three existing migration approaches: *source-based, destination-based, and hybrid (both source and destination).*

Source-based approaches choose the source to own the data during migration, and thus all client read and write queries are served by the source [22, 23, 53]. The source node iteratively migrates "dirty data" (data in the source that are already migrated but later updated) to the destination, which transfers additional data. Although source-based approaches can serve client queries without adding query latency, they have to terminate the source server at some point to copy the last piece of dirty data to the destination, incurring unavoidable service pauses. Source-based approaches have low query latency, but long migration time because migration operations compete with client queries at the source node.

Destination-based approaches choose the destination server to hold data ownership and serve client queries [28, 37]. All read and write queries will be routed to the destination. To serve data that have not yet migrated, the destination needs to pull the data from the source, and the client will have to retry after a wait. Therefore, destination-based solutions incur high query latency on not-yet-migrated items, especially at the beginning of migration, because most of the data are still on the source node. Meanwhile, destination-based approaches usually migrate data faster than source-based ones due to more resources available at the source.

Hybrid approaches (e.g., [30, 41]) can choose the destination node to handle write queries, and send read queries of not-yet-migrated and on-the-fly data to both source and destination nodes to achieve data consistency. Hybrid approaches need to keep track of migration process somewhere. For example, Fulva [30] tracks completed migration ranges in their key-value store client libraries. This type of approaches avoids on-demand data transfer between the source and the destination but instead uses additional network bandwidth (due to two read packets). Double-read incurs large resource overheads ($\sim 50\%$) among clients and two storage nodes to guarantee the protocol consistency because there is no fine-grained migration status tracking. Clients see the reply results from the node with a newer version and thus it can increase the latency by waiting for two replies.

Summary. Table 1 summarizes the performance characteristics and strengths/weaknesses of existing data migration protocols. We posit that data migration tasks usually have upper resource limits, and thus foreground client queries should be put on a higher priority than migration in the storage cluster. All existing live migration approaches have performance degradation and trade-off between migration cost and migration time. Our NetMigrate design provides a new alternative to improve these dimensions and reevaluate the trade-off between performance and migration time. By comparing the migration protocols, hybrid approaches do provide better migration performance compared with simply destination-based or source-based approaches, as they reduce the number of queries going to the wrong nodes.

Opportunities of programmable switches. Our aim is to design a switch-accelerated hybrid migration approach. A ToR switch positions centrally in all inter-server communications and acts as the gateway to other racks. This allows it to see all the migration and query traffic, enabling migration status tracking without additional communications. Host-based alternatives typically require sending migration status to a specific location (e.g., clients as in Fulva) or dealing with multiple requests going to the wrong place and pulling from another (e.g., Rocksteady or Slacker/Remus). Unlike CPUs, most programmable switches (e.g., Broadcom [16], Juniper [5], Intel [7]) are ASIC-based and offer flexible programmability without performance loss when performing customized modules. They can also guarantee high line rates such as 12.6 Tbps, orders-of-magnitude higher throughput and lower latency than servers. Therefore, deploying migration indexing service on switches can alleviate clients' or cluster coordinators' bookkeeping overheads when using hybrid approaches.

3 NetMigrate Overview

System architecture. NetMigrate is a rack-scale key-value store live migration accelerator leveraging in-network programmability. NetMigrate enables ToR switch as a migration state tracking service and routes client queries to the "right" server (either source or destination) during migration based on the latest information on the switch about what data have been completed migration or under migration. Fig. 1 shows the overall architecture of NetMigrate, which consists of a ToR switch, a controller, clients, and servers:

- **ToR switch** provides the following functionalities for the live migration service: (1) a *migration state table* module tracks migration states of each group of key-value pairs, indicating the data ownership belongs to the source or the destination. It uses probabilistic data structures and serves as an indexing service to determine where the client queries should go (§ 4.2); (2) a *routing* module routes client queries to the "right" storage server under migration (best-effort) based on the migration state table (§ 4.3); and (3) a *migration instance table* module records multiple pairs of key-value stores that are under migration for enabling (re-)routing client queries to the right corresponding storage.
- **Storage servers** store key-value data and serve client queries. In NetMigrate, we consider migration can occur between multiple storage instances within the same rack. Storage servers host key-value stores and run a migration agent that (1) maps key-value store API to NetMigrate migration packets, (2) serves client queries with consistency guarantees, and (3) handles data transmission between migrating storage instances. The migration agent makes NetMigrate easy and general to integrate with different backend key-value stores.
- **Clients** issue storage queries. NetMigrate provides a client agent to support the switch-based migration system. The client agent maps queries (e.g., `GET, SET, DELETE` commands) to the switch-based query packets, and transform NetMigrate reply packets into the backend storage commands. Migration process is transparent to client applications.

Challenges and Key Insights. To realize NetMigrate, we need to address several key design challenges:

- ***Fine-grained migration state tracking with limited on-switch resources.*** There is a disconnect between the potentially large key-value data to migrate and limited on-switch resources (e.g., SRAM, TCAM, etc.) that can be used to track migration status. It is infeasible to record the status of every single key. Therefore, we combine a number of key-value pairs together as a *group* and and record migration state at the group level. However, there is a tradeoff here: a too large group size (i.e., a small number of groups) limits the switch's ability to accurately determine the right destination for many queries, thereby compromising performance benefits, and vice versa. We need a memory-efficient design on the switch to support a large number of migration groups and the group sizes are relatively small. NetMigrate specifies three group-level migration states, *not-yet-migrated*, *ongoing-migration*, and *complete-migration*, to support fine-grained routing operations for client queries. At a high level, we use probabilistic data structures – Bloom filters (BF) [25], counting Bloom filters (CBF) [25, 29], and the hybrid of the two. We choose BF and CBF because of their memory efficiency and the fact that they can cover the required state tracking. BF tracks migrated keys at group-level and once a membership of a key is inserted, it cannot be removed. CBF tracks only ongoing-migration keys because we can delete keys from it once the keys are migrated. Hybrid of the two can indicate not-yer-migrated state. This choice leads to memory efficiency and simplistic design because Bloom filters can probabilistically perform membership tracking and involve only hashes and simple arithmetic operations.

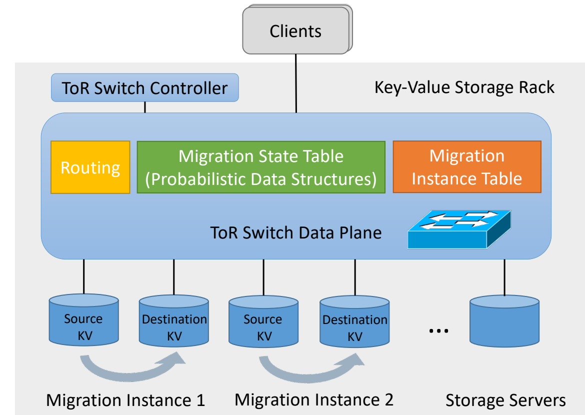


Figure 1: **NetMigrate system architecture.**

- ***Maintaining data consistency during migration.*** We consider the linearizability requirements [31] in consistency, including Read-After-Write (RAW), Write-After-Write (WAW), and Write-After-Read (WAR). To ensure consistency, it is critical to understand the right location that holds a key's up-to-date value and route the new queries to it during migration. The consistency issue becomes more challenging when our migration state tracking brings additional errors (e.g., false positives in BF and CBF). We propose a fine-grained error handling method to ensure correct query results as in § 4.3. At a high level, the key principle is that **we always route write queries to the destination unless we are sure that the migration has yet started, and issue read queries to both locations when we are definitely unsure about the migration state.** When we are *almost* confident that the data are on the destination but can have false positives, e.g., BF shows positive (the group has migrated) and CBF shows not positive (not under migration), we will route read queries to destination (and issue data pulls to the source if errors) instead of double

reads because the false positive rates are relatively small and we can reduce the workload for the source.

- ***Dynamic migration policies.*** One disadvantage of using existing live migration approaches is they have to sacrifice on or optimize towards a fixed set of dimensions (migration time or query performance). However, the operator may have different performance objectives when planning a migration (e.g., minimizing migration time or maximizing query performance) [43]. NetMigrate can optimize toward various performance goals (simulating any other protocols) and dynamically change the migration policy by tuning the probabilistic data structures in the switch and adjusting the CPU utilization of the source server. For example, we can simply set all BF entries to 1 to mimic destination-based solutions like Rocksteady [37]. To optimize migration time while offering better performance than source-based solutions, we can limit CPU usage to serve client queries and leave more CPU headroom for migration.

4 NetMigrate Design

In this section, we discuss the design of NetMigrate and describe how an in-network accelerator can help live migration.

4.1 Migration Workflow

Fig. 2 shows NetMigrate's general migration process. It starts from the current (source) server to a new server (destination) capable of accommodating the key-value shard. The source server initiates migration by notifying the destination server and registering the migration instance in the switch via *control packets* (Step ①). Throughout migration, clients remain unaware and continue sending queries to the original storage instances. The ToR switch decides whether a query should go to the source or destination based on migration status, and clients receive replies as if they are from the source server. During migration, the source server concurrently migrates data to the destination using *data packets*. The switch tracks the migration at the key-value *group* level, with each *group* containing multiple key-value pairs (Step ②). The destination server receives and replays these packets into its storage structures. Upon completion of the migration process, the source server agent issues a termination notification to the destination, switch, and clients (Step ③). In response, the source server cleans up its database, the switch removes the migration pair registration, the destination server takes over data ownership, and clients direct queries to the destination.

4.2 Migration State Tracking

NetMigrate accelerates key-value store live migration by tracking the migration states entirely in a central position (in the network) and using this in-network information to determine where to route the user queries as precisely as possible. However, it's impossible to store every migrated key given the large key space. Therefore, we make two design choices to shrink the tracking space requirement: (1) a combination

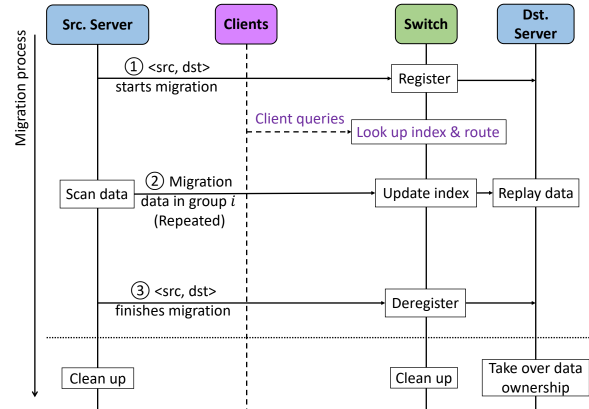


Figure 2: **NetMigrate migration workflow.**

of probabilistic data structures such as Bloom filers (BF) and counting Bloom filters (CBF) to track migration states with low false positives and fine-grained migration state, and (2) recording data ownership at a coarse granularity of *groups*. Each migration *group* represents a set of adjacent entries in the underlying key-value storage, e.g., several adjacent buckets in the hash table. These enable NetMigrate to scale to a large number of key-value pairs while maintaining high accuracy and low resource overhead.

Hybrid Bloom filters. The combination of BF and CBF is used to track three migration states: (`S1`) Entire group has not started migration; (`S2`) The group is under migration and only a subset of key-value pairs maybe migrated; (`S3`) Entire group has completed migration. Once a group (of key-value pairs) has completed migration, this group is recorded in BF. CBF tracks the "ongoing-migration" groups. When a group has started migration, this group is added to the CBF until migration is done. Compared to using a single BF to track which group(s) have completed migration, our hybrid design with both BF and CBF provides more fine-grained migration states and reduces false positives.

Specifically, when a group starts migration, the corresponding CBF entries are incremented by 1; when the group finishes migration, the same CBF entries will be deducted by 1. The states of each migration group are updated by its migration control packets – `GROUP-START-MIGRATION` and `GROUP-COMPLETE-MIGRATION` packets in the switch data plane (detailed packet format description is in Appendix A). Compared to using a single BF with the same memory space, NetMigrate's hybrid filters significantly reduce the false positives rate that can lead to throughput and latency degradation. We configure BF and CBF based on the following. For 2^x groups and 2^y migration parallelism (i.e., there are 2^y threads in total migrating key-value instances), the false positive rate upper bound of combining BF and CBF together is approximately $1-(1-(1-e^{\frac{-kn}{m}})^k)(1-(1-e^{\frac{-k'n'}{m'}})^{k'})$, where k and k' are the number of hash functions, n and n' are the total number of groups (elements), and m and m' are the number of entries used in BF and CBF respectively [25]. Here we

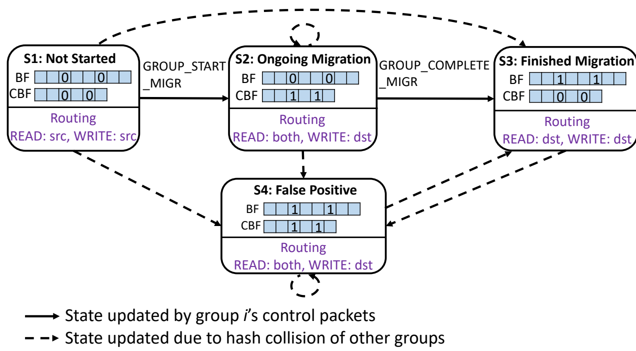


Figure 3: **NetMigrate migration state tracking and query handling state machine.** *Assuming two hash functions are used in BF and CBF, the values in BF and CBF arrays are located at the hashed indices of a particular group.*

consider the CBF's false positive is the same as BF's and use $k = k' = 4$ for both BF and CBF. If the target combined false positive rate is less than 1%, a good-enough configuration is using 2^{x+4}-bit Bloom filter and counting Bloom filter with 2^{y+4} entries (8 bits per entry) to achieve less than 0.5% combined false positive rate, resulting 16 bit-per-element space complexity. Also, 2^{x+3}-bit BF and 2^{y+3}-entry CBF (about 8 bits per element) can achieve a 5% false positive rate.

Group size tuning. The number of keys in a group is also a critical parameter that affects performance during migration. When the group size is smaller, the pending period where the switch cannot determine the location of the key is shorter. During the pending state (state (`S2`)), NetMigrate has to handle the wrong locations, adding up the performance degradation. In practice, there is an upper bound of the group size under which the performance impact is acceptable. As evaluated in § 6.4, the upper bound is around 2^{20} keys in a group. The lower bound of group size (namely, the upper bound of the number of groups given the total number of keys) is bounded by the BF and CBF sizes that the data plane can offer. Considering BF, CBF sizes and group sizes together, theoretically, 64 MB SRAM on switch will be able to support up to a 34-billion-key storage cluster migration in the same rack.

4.3 Data Consistency During Query Handling

It is important to avoid introducing additional inconsistency to the backend key-value stores during migration. As guaranteed by existing migration protocols, we consider the strongest data consistency (linearizability) [31] when designing our migration protocol – including Read-After-Write (RAW), Write-After-Write (WAW) and Write-After-Read (WAR) when handling client `READ` and `WRITE` queries. A migration state machine in Fig. 3 demonstrates the migration index tracking and query routing decisions in the switch data plane and to show how our protocol guarantees consistency during migration.

Migration packets updating indexes. Recall that there are three group migration states (`S1`-`S3` in Fig. 3). However, due to probabilistic errors and hash collisions in BF and CBF, there is a fourth migration state – false positive (`S4`). `S4` indicates a false positive because the BF and CBF entries of one group cannot be both positive simultaneously. We define migration control packets `GROUP-START-MIGRATION` to inform the switch that a specific group starts migration, and `GROUP-COMPLETE-MIGRATION` to notify the completion of the migration for a group. When a `GROUP-START-MIGRATION` packet arrives at the switch, it increments the CBF entries by 1, indicating a new ongoing migration. This transits state `S1` to `S2`. When a `GROUP-COMPLETE-MIGRATION` packet arrives, it sets the BF entries as 1 and decrements the corresponding CBF entries by 1, indicating that the group has finished migration. Thus, this action transits state `S2` to `S3`. Other transitions shown in Fig. 3 are false positives caused by hash collisions with other groups.

Query routing based on migration status look-up. Each state in Fig. 3 also shows index look-up results from BF and CBF and outlines query routing decisions. The general principle is: In the state machine, no state returns to a state with `READ` queries from the source (i.e., state `S1`). If a state `WRITE` to the destination, all possible following states are `READ` from either the destination or both nodes, ensuring access to the latest data. The states are explained as follows:

State `S1` means the group does not start migration, so both `READ` and `WRITE` queries are directed to the source. There are no false positives because both BF and CBF entries are 0.

State `S2` means the group is currently migrating. For `READ` query, the switch cannot determine whether the queried key-value pair is still on the source server, or on-the-fly, or on the destination server because the migration tracking granularity is larger than the per-key level. In this case, the switch generates double-read query packets via packet mirroring, where the original query is still forwarded to the source and a mirrored query is sent to the destination server. Thus, the client will receive two `READ` reply packets for this one `READ` query and merge two `READ` replies. If the destination has a successful reply, the client ignores the reply from the source because the destination may have updated values; otherwise, the key has not been migrated to the destination, so the client takes the source reply. We route the `WRITE` query to the destination in this state because the `READ` queries are doubled to both source and destination servers, and the client can always read the latest value. There can be false positives from CBF in this state, which are handled by double reads. The double-read ratio is low because the period when a group is under migration usually does not last long.

State `S3` stands for the group that finishes migration. Both `READ` and `WRITE` queries can be directed to the destination. There can be false positives from BF, and `READ` queries will be falsely directed to the destination while the data have not been migrated. In this case, NetMigrate agent on the destination issues PriorityPulls on-demand [37] to retrieve the missing key-value pairs from the source and respond to the client. This step also corrects the false positives of BF for subsequent queries as the keys are already at the destination.

State S4 represents a false positive case because a group cannot be in both a complete-migration and an ongoing-migration state simultaneously. To correct the false positive, we also use double-read for READ queries and direct WRITE queries to the destination.

Handling corner case. There is a corner case that some migrated key-value pairs can still be updated by WRITE queries in the source node. This happens when the WRITE query updates a key-value pair that is about to migrate. Specifically, this case occurs when a WRITE query arrives at the switch first and looks up the migration index. The index indicates that the key-value pair is still in the source node, and the switch forwards this query to the source. Next, the GROUP-START-MIGRATION packet is sent from the source, arrives at the switch, and updates the migration index. At the source, the data migration packet containing this key-value pair has been sent to the destination before the WRITE is executed. To guarantee data consistency in this case, we collect the late updates in memory at the source and periodically transfer the late dirty logs to the destination as a source-based protocol.

In summary, NetMigrate can ensure data consistency during migration. Moreover, experiments in § 6.4 show a low overhead to correct false positives, with the portion of double reads and PriorityPulls being less than 0.05%, and less than $4 \times 10^{-5}\%$ extra bandwidth overhead incurred by the late dirty logs.

4.4 Dynamic Migration Policies

There is a fundamental tradeoff between migration time and the query performance: The migration completion time is related to the source's CPU headroom left for migration, while the query performance also depends on the source's and destination's CPU cycles for query serving. By configuring migration CPU cycles and taking advantages of Bloom filters, NetMigrate has more flexibility to be configured to support various migration goals, such as minimizing migration time or optimizing query throughput and latency.

- **Minimize migration time.** As shown in the experiments, Rocksteady has the shortest migration time because all the source CPU cycles can be used for migration. To achieve the similar migration time as Rocksteady, one way is to simply pre-set all BF entries as 1, indicating that all queries should be routed to the destination. NetMigrate will issue PriorityPulls to fetch not-yet-migrated keys. Thus, NetMigrate's protocol is now essentially the same as that of Rocksteady. However, this strawman solution only gives NetMigrate the same query performance as Rocksteady. Alternatively, we can limit the CPU cycles for serving client query in the source to a *low* level and leave more CPU headroom for migration. By doing so, NetMigrate achieves a similar migration time as Rocksteady, while gaining some throught and latency benefits because BF correctly identifies the keys belonging to the source.
- **Maximize query throughput and minimize latency.** NetMigrate is designed to gain more benefits in query performance from the source and destination. To maximize query throughput and minimize query latency, we set the CPU cycles in the source for client queries to a *medium* level, and leave some CPU headroom for migration. By doing so, NetMigrate achieves the highest throughput and lowest median latency compared to other baselines while maintaining a similar migration time, as detailed in §6.2.
- **Mimic other migration protocols and take advantages of all.** An interesting feature of NetMigrate is that it can be configured to hybrid and source-based protocols in addition to Rocksteady because its design takes fine-grained migration states into consideration. To make it equivalent to Fulva (hybrid protocol), we can pre-increment all CBF entries by 1 so that it will be in state S2 or S4 forever. To make it the same as a source-based protocol, NetMigrate needs to disable the BF and CBF updates, which keeps its state in S1. NetMigrate also consists of transferring late dirty logs from the source to the destination, similar to a source-based protocol. In addition to these, we observe in the evaluation (§ 6.3) that a medium-size BF and CBF can give the best query performance, e.g., the 8-bit-per-element setting in Table 2, compared to the ones with more BF and CBF space. This is because some false positives in the switch index actually shift the query workload from the source to the destination, which gives more CPU headroom for the source to migrate data and helps move the workloads to destination faster. Thus, by adjusting some false positives of BF and the headroom of the source CPU for migration, NetMigrate can take the performance advantage of both destination-based and source-based approaches while maintaining data consistency.

5 Implementation

We have implemented a prototype of NetMigrate with Redis [11] as an example, including the programmable data plane serving as a migration index, the migration server agents, and the client running YCSB workloads. The indexing switch is implemented with 2K lines of P4-16 code and is compiled to Intel Tofino ASIC [7]. We implement the migration instance table using a P4 table and the migration state table using BF and CBF where each BF entry is 1 bit and each CBF entry has 8 bits. Both BF and CBF use 4 different hash functions. We implement L3 routing and redirect client queries by changing their destination or mirroring queries to both storage nodes. The switch control plane is implemented with 600 lines of Python code, which registers and deregisters the migration instances by modifying the migration instance table in the control plane. The migration server agents and clients are implemented with 15K lines of C++ code. In the prototype, we use the Redis-plus-plus library [14] to communicate with Redis instances in migration servers. We add three new User-Defined Functions to get the current hash ta-

ble information for migration and to scan key-value pairs in the order of hash values. Source server agents call the user-defined commands, scan key-value pairs in parallel, and send migration control and data packets via UDP sockets. Destination server agents receive migration packets and insert data into the destination instance. We modified the C++ YCSB client [17] for key-value store and UDP communication.

6 Evaluation

We conduct extensive experiments comparing NetMigrate to the latest live migration solutions and demonstrate:

- NetMigrate improves the query throughput by 6.5% to 416% under different YCSB workloads and load-balancing scenarios (§6.2 and §6.5) while keeping low tail latency.
- NetMigrate supports diverse migration policies with different performance goals, including query throughput, latency, and migration completion time. It improves the average query throughput from 32% to 78% compared to baseline protocols with similar migration time (§6.2).
- NetMigrate can achieve the above improved performance with slim BF and CBF space allocation within the switch memory limitation (§ 6.3).
- NetMigrate incurs negligible bandwidth overhead (§6.4).

6.1 Methodology

Testbed. The experiments are conducted on a testbed consisting of one 6.5 Tbps Intel Tofino switch and 3 commodity servers. Each server is equipped with an 8-core CPU (Intel Xeon E5-2620 @ 2.10GHz), 64GB total memory (two 32GB DRR4-2400 DRAMs), and one 40G NIC (Intel XL710).

KV workloads. By default, we create Redis key-value stores consisting of 256 million key-value pairs (∼16GB), occupying 52.7% memory of a server (33GB including Redis indexing data structures), with 4-Byte keys and 64-Byte values. We use YCSB benchmark [26] designed for key-value stores evaluation. The queried keys are generated randomly according to the Zipfian distribution with $\theta = 0.99$. We use 5% write ratio and 100% source Redis CPU usage budget to show the overall performance impact in § 6.2. We furthertune the workload write ratio among 0% (YCSB-C), 5% (YCSB-B), 10%, 20%, and 30%, and limit source Redis CPU to different budgets, i.e., 100% (not overloaded), 70% (slightly overloaded), and 40% (heavily overloaded), using `cpulimit` [3] to create different load balancing scenarios in § 6.5.

Evaluation parameters and metrics. By default, we set BF size to 512 KB and CBF size to 1024 KB, with which uses 4 hash functions. The default CBF size of 1024KB was a sufficiently large starting point as we were not sure how many KV pairs are on-the-fly during migration. Additional sensitivity tests in Table 2 show that 1024KB CBF is usually an overkill but it's significantly smaller than the total switch memory. We configure 2^{17} migration groups, each of which has up to 2048 key-value pairs. In the experiments, we show client-observed performance metrics, such as Queries per Second (QPS), end-to-end latency, and migration completion time. We use extra bandwidth percentages between a client and servers (denoted as client-size), and between the source and destination servers (denoted as server-size) to evaluate the extra migration overhead.

Baselines. We implement three types of migration systems and their protocols (as discussed in §2.2) in our testbed for a fair comparison. All baselines follow the same data I/O and network protocols, export the key-value pairs from the source, and use the migration agents at both the source and the destination to transmit the key-value data, as shown in Fig. 2. The difference is that they do not use switch indexing. We implement (1) *source-based protocols* including Slacker [23] and Remus [33]; (2) *destination-based protocols* including Rocksteady [37] with gRPC asynchronous API [6] to implement the PriorityPull RPCs; and (3) *hybrid protocols* including Fulva [30]. In particular, the client in Fulva keeps track of the migration progress and the hot keys are migrated with a higher priority based on sampling statistics.

6.2 Overall Performance

In this experiment, we consider a migration scenario where both the source and the destination are not overloaded and have 100% CPU budgets for Redis. We show the client-side throughput and latency during migration using YCSB-B workloads with a 5% write ratio.

Query throughput. This experiment highlights the throughput difference and performance trade-offs in different migration protocols. Fig. 4 (a), (b) and (c) show the throughput and migration time of the three baselines. Compared to the three baselines, NetMigrate improves the average query throughput by 78.2%, 56.5%, and 31.9% when it is configured as high, medium, and low migration speeds respectively. Among the baselines, (1) Rocksteady has the lowest throughput and also the shortest migration time because all queries are handled by the destination Redis, leaving the most CPU headroom for source Redis to perform migration. (2) Fulva's client throughput has been cut by a half compared to a fully-utilized Redis instance's throughput because of the overhead caused by double-reads and being bounded by the packet rate. (3) Source baseline's throughput keeps stable during migration and it is slightly lower than without migration because it uses the smallest portion of source Redis CPU for migration and thus its migration time is the longest.

When zooming into NetMigrate, we can see that NetMigrate's throughput first increases to a peak level and then drops as depicted in Fig. 4 (d), (e), and (f). This is because at the beginning of migration, client queries are mainly served by the source. During migration, increasing numbers of key-value pairs are migrated to the destination. The destination Redis can serve queries of the already-migrated data, and thus total query throughput increases. When most data are migrated to the destination, destination is pressured from both

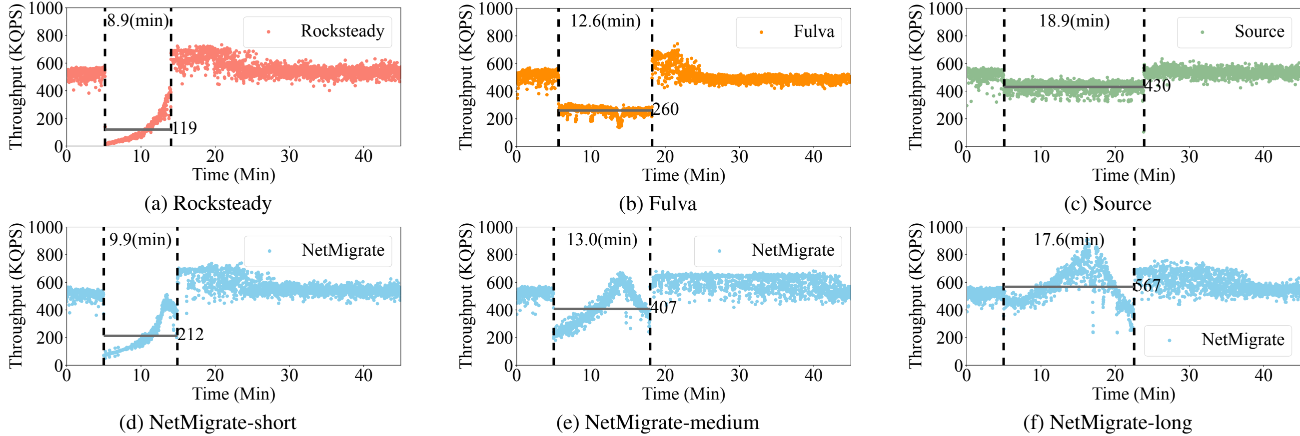


Figure 4: **Comparing Rocksteady, Fulva, Source-based protocol, and NetMigrate on query throughput with YCSB-B workload and 100% source CPU budget.** *We report the average throughput as the horizontal lines, label the migration start and finish timestamps as dotted lines, and show migration time of each protocol in the figures. NetMigrate-short, NetMigrate-medium, and NetMigrate-long denote the experimental results when NetMigrate is configured to different migration time policies.*

migrated data insertion and query serving, so the throughput drops when migration is nearing the completion and there is a peak throughput point during migration. The peak throughput is even larger than a single Redis' query throughput because the ToR switch lets the client leverage both the source and the destination Redis's query power.

Query latency. We then evaluate the latency percentiles as shown in Fig. 5 and 6. The average median latencies of Rocksteady, Fulva, and Source baseline are all larger than NetMigrate's under any migration time configurations. NetMigrate reduces the average median latency from 49% to 65% in all cases. For average 99%-tail latency, NetMigrate is better than Rocksteady and Fulva when it's configured to the similar migration time, reducing the latency by 27.0% and 39.5% correspondingly. NetMigrate-long's 99%-tail latency is almost twice than Source baseline's because in the worst case, NetMigrate still needs to wait for two replies from both source and destination Redis and it can have PriorityPulls for wrongly directed queries.

NetMigrate's adaptable migration policies. NetMigrate can adjust between migration cost and migration time based on the user needs. We limit source Redis client query processing CPU cycles to adjust the CPU headroom for migration, and NetMigrate can migrate data with high migration speed (similar as Rocksteady), medium migration speed (similar as Fulva), and low migration speed (similar but better than Source baseline) based on the configurations. Fig. 4, 5, 6 (d), (e), and (f) show NetMigrate is adaptable to different migration time requirements and demonstrates different query performance levels. Also, NetMigrate can achieve similar migration time while maintaining higher throughput and lower access latency compared to all three baselines (except for comparing to Source baseline in the case of 99%-tail latency).

6.3 Tuning Bloom Filter Sizes and Group Sizes

Bloom filter size tuning. We evaluate the impact of BF and CBF sizes on migration and query performance. We also run real migration experiments changing the BF and CBF sizes with totally 2^{17} migration groups and 4 threads to migrate in parallel. Table 2 shows that combining BF and CBF reduces false positives in practice significantly. The client-side extra bandwidth usage can reveal actual false positives. In Table 2, given a large enough BF size, we shrink the CBF sizes and find that 64 Bytes CBF is the tuning point before performance drops. Keeping the good-enough CBF size, we shrink the BF sizes. Results show that 8-bit-per-element gives the best performance while avoiding wasting too much space. When the actual false positive rate is too high, e.g., when space complexity is less than 2 bits per element, client-side extra bandwidth usage and 99%-tail latency are worse due to the increased number of PriorityPulls and double-reads to correct false positives.

Group size tuning. Given large enough BF and CBF sizes, e.g., 512KB BF and 64B CBF, we tune the group size (i.e., the total number of groups). Table 3 shows that when group size (i.e., the number of keys in the group) is larger than 2^{20}, the throughput and latency will be harmed because of the increased double-reads when the queried key is in an ongoing-migration group.

6.4 Extra Overhead for Migration

Extra bandwidth usage. Table 4 shows extra bandwidth usage under all write ratios and source CPU budget settings. Source protocol's extra bandwidth usage only comes from the server side, where the source server needs to transfer dirty logs to the destination when `WRITE` queries are later than the migration of their keys. Rocksteady's only comes from the client side, where client needs to retry queries with PriorityPulls. For Fulva, double-reads contribute to almost a

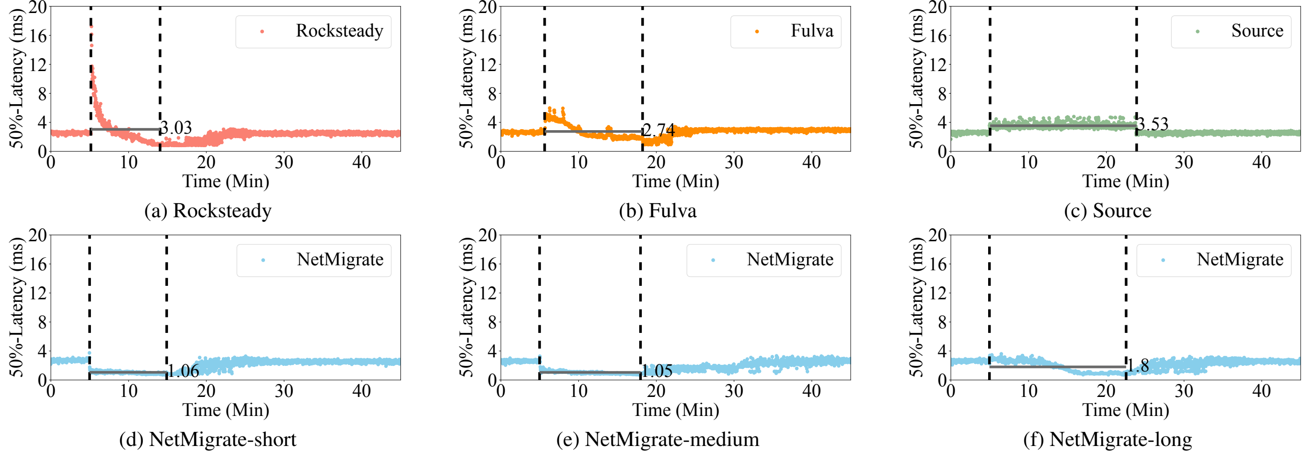


Figure 5: **Comparing Rocksteady, Fulva, Source-based protocol, and NetMigrate on median latency with YCSB-B workload and 100% source CPU budget.** *We report the average median latency as horizontal lines and label migration start and finish timestamps as dotted lines.*

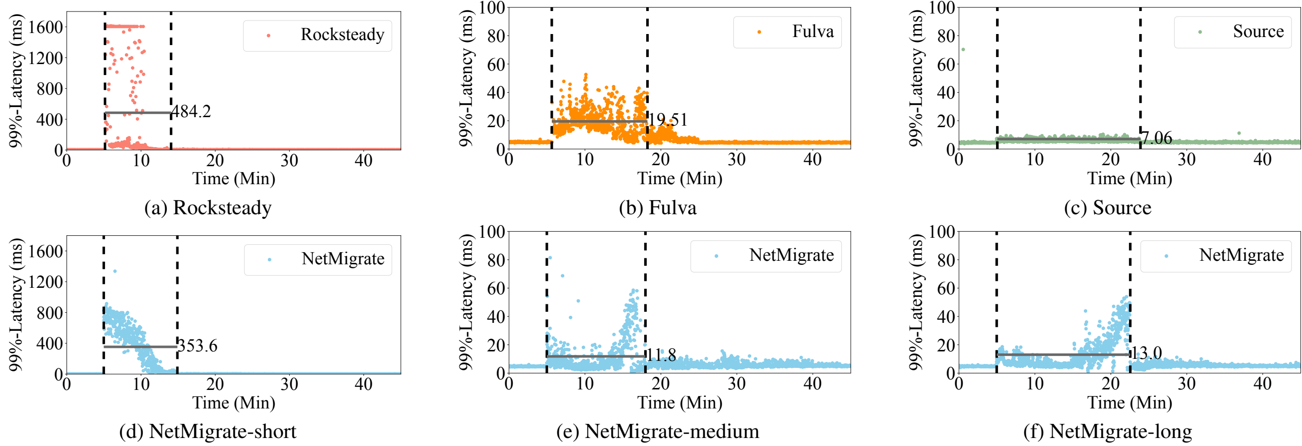


Figure 6: **Comparing Rocksteady, Fulva, Source-based protocol, and NetMigrate on 99%-tail latency with YCSB-B workload and 100% source CPU budget.** *We report the average 99%-tail latency as horizontal lines and label migration start and finish timestamps as dotted lines.*

BF, CBF size	FP	Bits/Ele	Throughput	Median, 99% Latency	Client BW
512 KB, 128 B	0.038%	32	564.1 KQPS	1.13 ms, 7.83 ms	0.06%
512 KB, 64 B	0.26%	32	572.7 KQPS	0.92ms , 7.23 ms	0.30%
512 KB, 32 B	2.42%	32	362.5 KQPS	0.89 ms, 26.5 ms	1.37%
512 KB, 16 B	16.0%	32	197.5 KQPS	1.15 ms, 190.2 ms	3.73%
256 KB, 64 B	0.48%	16	569.6 KQPS	0.94 ms, 5.61 ms	0.30%
128 KB, 64 B	2.63%	8	573.2 KQPS	0.93 ms, 4.47 ms	0.30%
64 KB, 64 B	16.2%	4	563.6 KQPS	0.95 ms, 5.42 ms	0.30%
32 KB, 64 B	56.0%	2	523.8 KQPS	0.93 ms, 4.38 ms	0.47%
16 KB, 64 B	92.9%	1	495.8 KQPS	1.01 ms, 5.73 ms	0.65%

Table 2: **Impact on migration when tuning BF and CBF sizes.** *"FP" represents the upper bound of combined false positive rates of BF and CBF. "Bits/Ele" stands for total BF and CBF bits per element. "Client BW" means the extra bandwidth usage between the client and servers, compared with total query traffic. In the settings listed in the table, the server-side extra bandwidth usages are all less than* 6×10^{-5}*% and negligible.*

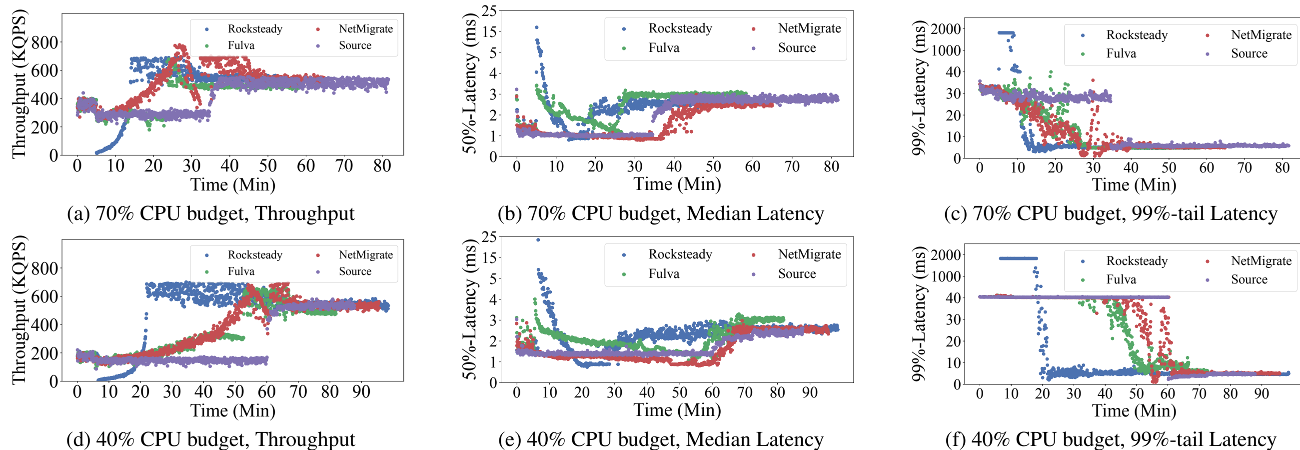
Group size	Throughput	Median, 99% Latency	Client BW
2^{11}	567 KQPS	2.33 ms, 8.46 ms	0.0060%
2^{14}	599 KQPS	1.05 ms, 5.19 ms	0.0165%
2^{18}	573 KQPS	0.95 ms, 10.97 ms	0.2613%
2^{19}	561 KQPS	0.91 ms, 3.82 ms	0.3733%
2^{20}	521 KQPS	0.9 ms, 3.63 ms	0.4858%
2^{21}	494 KQPS	0.9 ms, 4.01 ms	0.66%
2^{24}	255 KQPS	0.94 ms, 86.27 ms	2.76%
2^{25}	180 KQPS	1.07 ms, 250.52 ms	4.05%

Table 3: **Impact on migration when tuning group sizes.** *"Client BW" means the extra bandwidth usage between the client and servers.*

half of extra READ query packets in the client-side and there is no extra communication between servers. NetMigrate's extra bandwidth usage comes from both the client and server sides, but they are both negligible as shown in the results. The client-side extra usage comes from PriorityPull query retries as well as double-reads for undecidable conditions in the switch indexing; the server-side extra usage is from transferring late dirty logs to the destination as Source protocol. As shown in Table 4, Source protocol's extra bandwidth usage from the server side is proportional to the workload write ratio. Fulva's extra bandwidth usage all exceeds 35% and is several times higher than Rocksteady's. Rocksteady's and Fulva's client-side extra usages decrease with the write ratio increases because WRITE queries are directly served by the destination. NetMigrate achieves negligible extra bandwidth usage from both the client side and the server side. NetMigrate's client-

Write Ratio	0%						10%						20%						30%					
Source CPU Budget	100%		70%		40%		100%		70%		40%		100%		70%		40%		100%		70%		40%	
Extra Bandwidth Usage (%)	C	S	C	S	C	S	C	S	C	S	C	S	C	S	C	S	C	S	C	S	C	S	C	S
Source	0	0	0	0	0	0	0	10.5	0	10.2	0	9.8	0	20.7	0	20.6	0	20.5	0	31.3	0	30.9	0	30.6
Rocksteady	12.5	0	12.6	0	11.3	0	10.9	0	10.4	0	9.3	0	8.6	0	8.6	0	7.5	0	6.9	0	6.8	0	5.5	0
Fulva	58.1	0	60.9	0	50.3	0	49.4	0	55.0	0	46.6	0	43.6	0	58.2	0	53.4	0	36.9	0	39.2	0	44.1	0
NetMigrate (S $\times 10^{-5}$%)	0.025	3.11	0.031	3.39	0.05	2.65	0.005	3.27	0.005	2.69	0.005	1.05	0.004	1.36	0.004	3.74	0.004	2.37	0.003	1.71	0.003	2.88	0.003	2.1

Table 4: **Extra bandwidth usage between a client and servers and between two migration servers.** *'C' stands for Client-side extra bandwidth usage and 'S' stands for that of server-side. NetMigrate's server-side extra usage is at 10^{-5}% level.*



(a) 70% CPU budget, Throughput (b) 70% CPU budget, Median Latency (c) 70% CPU budget, 99%-tail Latency

(d) 40% CPU budget, Throughput (e) 40% CPU budget, Median Latency (f) 40% CPU budget, 99%-tail Latency

Figure 7: **Comparing Rocksteady, Fulva, Source-based protocol, and NetMigrate on throughput, median latency and 99%-tail latency with YCSB-B workload (5% write ratio), 70% and 40% source CPU budget.**

Write Ratio	0%			10%			20%			30%		
Source CPU Budget	100%	70%	40%	100%	70%	40%	100%	70%	40%	100%	70%	40%
Source	445.66	289.16	152.30	417.98	289.17	145.96	424.72	266.27	140.38	412.77	266.51	142.25
Rocksteady	113.17	114.52	67.73	119.11	125.15	77.53	133.76	132.34	84.07	147.79	145.57	99.28
Fulva	239.68	251.77	201.73	262.27	278.99	243.49	244.15	296.81	253.36	241.86	247.76	261.79
NetMigrate	**584.41**	**459.58**	**297.36**	**554.67**	**446.83**	**290.83**	**549.70**	**427.67**	**286.42**	**559.92**	**420.09**	**278.75**

Table 5: **Throughput under varied write ratios and source Redis CPU budgets.**

side extra usage indicates that double-reads issued by the switch and PriorityPulls from the destination happen less than 0.05%. Also, NetMigrate's server-side extra usage is less than 4×10^{-5} and negligible. Therefore, NetMigrate puts much less overhead to both the clients and the servers than other three baselines.

6.5 More Scenarios and Workloads

In this section, we evaluate more load-balancing scenarios and write-sensitive workloads by tuning the source Redis CPU limits and write ratios in the YCSB benchmark.

Load balancing scenarios. In a case that needs load balancing, the source node is usually overloaded and the destination node serves queries faster than the source. We mimic different overload levels by limiting the source Redis CPU to 70% and 40%. In this experiment, we configure NetMigrate to be throughput-optimized. Fig. 7 shows the throughput and latency comparisons among four migration protocols. Fig. 7 (a) and (d) show the throughput results. A lower source CPU budget for migration leads to a longer migration time for all protocols. NetMigrate's average throughput during migration are the highest for both 70% and 40% source CPU limitations. NetMigrate improves the throughput from 63% to 286% with 70% CPU limitation and from 29% to 305% with 40% CPU limitation. For Rocksteady, the less the CPU budget is given, the more slowly the throughput increases from the nearly zero QPS. For NetMigrate and Fulva, at the beginning of migration, the throughput improvement curves are similar, because both are limited by the migration speed. After that, NetMigrate is better than Fulva because our client is not bounded by the client-side packet rate. Source baseline keeps a low but stable throughput. Fig. 7 (b), (c), (e), and (f) show the latency results. For both 70% and 40% CPU budgets, median latency of NetMigrate and Source baseline remains low and stable, while Rocksteady's and Fulva's latency suddenly increases and gradually drops during migration. NetMigrate's median latency remains the lowest compared to other baselines. It reduces the median latency from 8% to 65% with 70% CPU limitation and from 32% to 97% with 40% CPU limitation. In terms of 99%-tail latency, Rocksteady is two orders of magnitude higher than other migration protocols during the entire migration due to its on-demand data fetching. However, it quickly falls to normal tail latency when migration finishes. Both NetMigrate's and Fulva's tail latencies drop gradually

Write Ratio	0%			10%			20%			30%		
Source CPU Budget	100%	70%	40%	100%	70%	40%	100%	70%	40%	100%	70%	40%
Source	3.38	**1.09**	1.40	3.65	1.23	2.38	3.55	1.31	1.33	3.62	1.21	1.28
Rocksteady	3.43	3.15	2.97	3.30	3.14	2.45	2.83	2.70	2.19	2.81	2.51	1.92
Fulva	3.11	2.07	1.93	2.58	2.10	1.80	2.72	2.09	1.80	2.79	2.38	1.82
NetMigrate	**2.12**	1.20	**1.10**	**2.26**	**1.11**	**1.05**	**2.27**	**1.05**	**1.08**	**2.20**	**0.97**	**1.05**

Table 6: **Median latency under varied write ratios and source Redis CPU budgets.**

Write Ratio	0%			10%			20%			30%		
Source CPU Budget	100%	70%	40%	100%	70%	40%	100%	70%	40%	100%	70%	40%
Source	**6.40**	29.32	62.11	**6.83**	30.05	66.01	**7.19**	31.03	68.22	8.11	30.31	63.29
Rocksteady	491.86	504.92	864.89	368.89	425.64	973.53	331.37	346.02	866.59	213.79	227.03	795.49
Fulva	21.75	23.34	48.75	21.04	24.59	42.36	18.51	23.42	46.06	21.50	22.65	38.79
NetMigrate	9.21	**23.09**	**48.31**	7.73	**21.74**	**33.89**	8.38	**19.19**	**41.28**	**7.50**	**15.89**	**34.75**

Table 7: **99%-tail latency under varied write ratios and source Redis CPU budgets.**

from the high latency level before migration while the tail latency of Source protocol remains the same as before migration until it is approaching migration completion. Overall, NetMigrate reduces 99%-tail latency from 18% to 56% with 70% CPU limitation and up to 94% with 40% CPU limitation.

Diverse write ratios. Changing the YCSB workloads among different write ratios (0%, 10%, 20%, and 30%), Table 5, 6, and 7 show that NetMigrate can achieve the highest throughput (improved from 6.5% to 416%) while maintaining the lowest latency as Source baseline. Rocksteady's 99%-tail latency is also much higher than other migration protocols when write ratios and source CPU budgets change.

7 Discussion and Related Work

Migration speed. In our experiments, migration time is limited by exporting key-values out from Redis and then sending through UDP socket. Key-value stores utilizing RDMA or other kernel-by-passing transmission (e.g., Intel DPDK [4], MICA [40], KV-Direct [39]) can increase migration speed by a lot. Despite kernel-bypassing, migration time remains non-negligible (e.g., 60 sec for 200GB data, 40Gbps links [37]). Migration degrades query performance significantly and migration happens fairly frequently in the storage clusters. NetMigrate can work with faster networking to improve the KV serving performance during migration.

Fault tolerance is also critical during migration, including server failures and switch failures. To handle server failures, enabling logs on both source and destination key-value storage servers is a viable solution. Recovery is achieved by merging logs from both sides to attain the latest version. RedPlane [35] and ExoPlane [36] provide fault-tolerant solutions for switch failures and resource augmentation.

Strong/weak data consistency. UDP-based protocol can have packet loss and out-of-order transmission, which weakens the data consistency. We can add a reliable transmission mechanism to our UDP-based migration protocol, and thus it can be robust to give a strong data consistency over network transmission. The migration control packet replies are generated by the switch and sent back to the source node. This avoids duplicated updates in the switch index structures. Also, when NetMigrate merges data insertion from migration and write queries at the destination, it needs the key-value store's version numbers to guarantee strong consistency. In practice, many key-value stores provide weak consistency and sacrifice consistency for availability and performance [12, 19]. NetMigrate is compatible with weak data consistency.

Key and value sizes and multi-key operations. We use 4-Byte keys and 64-Byte values in the prototyping experiments but NetMigrate can be extended to larger key and value lengths as long as a group id and a single key can be fit into switch metadata (128 bits at most). If the key size is relatively small, we can also extend the packet format to support multi-key operations in one packet and recirculate one query packet in the switch and treat each pass as serving a single-key query.

Clearing bloom filters in practice. When a shard of data migration information is updated to the storage cluster indexing proxy, the cluster scheduler can pause migrations for a bit (for synchronization) and clean the probabilistic indexing data structures shared in the switch periodically.

Migration plan. There are works generating reconfiguration plan based on cluster load status, migration time, performance impact and so on [18, 21, 41, 43, 44, 47, 48, 50], while NetMigrate focuses on live migration technique.

8 Conclusions

We present NetMigrate, a new live migration approach for in-memory key-value stores based on programmable data planes. NetMigrate migrates shards between nodes with zero service interruption and minimal performance impact using switches for migration status tracking. Extensive experimental results demonstrate the ability of NetMigrate to provide enhanced throughput and maintain low access latency under a variety of changing workloads and scenarios during migration.

Acknowledgments. We would like to thank the anonymous reviewers and our shepherd Jiri Schindler for their helpful comments. We thank Zhuolong Yu for help with the testbed setup and debugging. This work was supported in part by NSF grants CNS-2107086, SaTC-2132643, CNS-2106946.

References

[1] Apache Cassandra. https://cassandra.apache.org.

[2] Building a Gigascale ML Feature Store with Redis, Binary Serialization, String Hashing, and Compression. https://doordash.engineering/2020/11/19/building-a-gigascale-ml-feature-store-with-redis/.

[3] CPU limit tool. https://github.com/opsengine/cpulimit.

[4] DPDK. https://www.dpdk.org/.

[5] EX9200 Line of Ethernet Switches. https://www.juniper.net/us/en/products/switches/ex-series/ex9200-programmable-network-switch.html.

[6] gRPC. https://grpc.io/.

[7] Intel Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[8] Memcached. https://memcached.org.

[9] NetMigrate key-value store live migration codebase. https://github.com/Froot-NetSys/NetMigrate.

[10] P4 Language. https://opennetworking.org/p4/.

[11] Redis. https://redis.io.

[12] Redis Cluster consistency guarantees. https://redis.io/docs/management/scaling/.

[13] Redis MIGRATE command. https://redis.io/commands/migrate/.

[14] Redis-plus-plus library. https://github.com/sewenew/redis-plus-plus.

[15] Redis use cases. https://redis.com/blog/5-industry-use-cases-for-redis-developers/.

[16] Trident 5 / BCM78800 Series. https://www.broadcom.com/products/ethernetconnectivity/switching/strataxgs/bcm78800.

[17] YCSB Benchmark written in C++. https://github.com/ls4154/YCSB-cpp.

[18] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 739–753, USA, 2016. USENIX Association.

[19] Eric Anderson, Xiaozhou Li, Mehul A Shah, Joseph Tucek, and Jay J Wylie. What consistency does your {Key-Value} store actually provide? In *Sixth Workshop on Hot Topics in System Dependability (HotDep 10)*, 2010.

[20] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. Sharding the shards: managing datastore locality at scale with akkio. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 445–460, 2018.

[21] Saurabh Bagchi, Somali Chaterji, Paul Curtis Wood, and Ashraf Mahgoub. Clustered database reconfiguration system for time-varying workloads, September 6 2022. US Patent 11,436,207.

[22] Sean Barker, Yun Chi, Hakan Hacigümüs, Prashant Shenoy, and Emmanuel Cecchet. {ShuttleDB}:{Database-Aware} elasticity in the cloud. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 33–43, 2014.

[23] Sean Barker, Yun Chi, Hyun Jin Moon, Hakan Hacigümüş, and Prashant Shenoy. " cut me some slack" latency-aware live migration for databases. In *Proceedings of the 15th international conference on extending database technology*, pages 432–443, 2012.

[24] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[25] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[26] 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*, pages 143–154, 2010.

[27] 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. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[28] Aaron J Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 299–313, 2015.

[29] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on networking*, 8(3):281–293, 2000.

[30] Jiewen Hai, Cheng Wang, Xusheng Chen, Tsz On Li,

Heming Cui, and Sen Wang. Fulva: Efficient live migration for in-memory key-value stores with zero downtime. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 83–8309. IEEE, 2019.

[31] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[32] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, 2017.

[33] Junbin Kang, Le Cai, Feifei Li, Xingxuan Zhou, Wei Cao, Songlu Cai, and Daming Shao. Remus: Efficient live migration for distributed databases with snapshot isolation. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2232–2245, 2022.

[34] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. Zeus: Locality-aware distributed transactions. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 145–161, 2021.

[35] Daehyeok Kim, Jacob Nelson, Dan RK Ports, Vyas Sekar, and Srinivasan Seshan. Redplane: Enabling fault-tolerant stateful in-switch applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 223–244, 2021.

[36] Daehyeok Kim, Vyas Sekar, and Srinivasan Seshan. {ExoPlane}: An operating system for {On-Rack} switch resource augmentation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1257–1272, 2023.

[37] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 390–405, 2017.

[38] Herman Lee Kwiatkowski, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, and David Stafford. Scaling memcache at facebook.

[39] 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*, pages 137–152, 2017.

[40] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. {MICA}: A holistic approach to fast {In-Memory}{Key-Value} storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, 2014.

[41] Yu-Shan Lin, Shao-Kan Pi, Meng-Kai Liao, Ching Tsai, Aaron Elmore, and Shan-Hung Wu. Mgcrab: transaction crabbing for live migration in deterministic database systems. *Proceedings of the VLDB Endowment*, 12(5):597–610, 2019.

[42] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. {DistCache}: Provable load balancing for {Large-Scale} storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, 2019.

[43] Chenyang Lu. Aqueduct: Online data migration with performance guarantees. In *Conference on File and Storage Technologies (FAST 02)*, 2002.

[44] Ashraf Mahgoub, Paul Wood, Alexander Medoff, Subrata Mitra, Folker Meyer, Somali Chaterji, and Saurabh Bagchi. SOPHIA: Online reconfiguration of clustered NoSQL databases for Time-Varying workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 223–240, Renton, WA, July 2019. USENIX Association.

[45] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[46] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015.

[47] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *Proceedings of the VLDB Endowment*, 7(12):1035–1046, 2014.

[48] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *Proc. VLDB Endow.*, 10(4):445–456, nov 2016.

[49] Ravindra Kumar Singh and Harsh Kumar Verma. Redis-based messaging queue and cache-enabled parallel processing social media analytics framework. *The Computer Journal*, 65(4):843–857, 2022.

[50] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained

elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3):245–256, nov 2014.

[51] Bo Wang, Changhai Wang, Ying Song, Jie Cao, Xiao Cui, and Ling Zhang. A survey and taxonomy on workload scheduling and resource provisioning in hybrid clouds. *Cluster Computing*, 23:2809–2834, 2020.

[52] Ke Wang, Xraobing Zhou, Tonglin Li, Dongfang Zhao, Michael Lang, and Ioan Raicu. Optimizing load balancing and data-locality with data-aware scheduling. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 119–128. IEEE, 2014.

[53] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. Replication-driven live reconfiguration for fast distributed transaction processing. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 335–347, 2017.

A NetMigrate Network Protocol

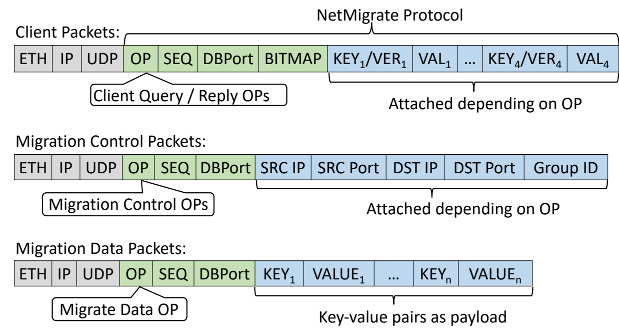


Figure 8: **NetMigrate packet format for migration packets and client query/reply packets.**

Packet format. Fig. 8 shows the packet format in NetMigrate protocol. NetMigrate is an application-layer protocol inside L4 payload. A set of dedicated UDP ports are reserved for key-value storage server/client agents. In the switch, these ports indicate the packets are NetMigrate migration packets, client query, or reply packets, to invoke the custom packet processing logic. The NetMigrate header fields are `OP`, `SEQ`, and `DBPort`. `OP` fields represents client query or reply operators, or migration-related operators. `SEQ` can be used as a sequence number for reliable transmissions with UDP protocol. `DBPort` refers to which key-value store instance the packet is responsible for, filled with the application port. For packets deal with client queries and data migration, they have `KEY`, `VALUE`, or `VERSION` fields. `KEY` and `VALUE` carry the key and value of a key-value pair and `VERSION` indicates the version number in reply packets for data consistency guarantees during migration. NetMigrate supports `GET`, `SET`, and `DELETE` client query types and can be extended to other type of queries. `GET` and `DELETE` query operators only have `KEY` fields; `SET` packets have both `KEY` and `VALUE` fields; and all reply packets has `VERSION` fields indicating whether the operation is successful and the reply is from the source or the destination storage instance. The header fields for migration control packets are `OP`, and migration instance `(SRC_IP, SRC_Port, DST_IP, DST_Port)` or `group_id` which is attached depending on the `OP`. `OP` can be `MIGRATE_INIT`, `MIGRATE_TERMINATE`, `MIGRATE_GROUP_START`, `MIGRATE_GROUP_COMPLETE`, `MIGRATE_DATA`, and their corresponding reply operators. `MIGRATE_INIT` and `MIGRATE_TERMINATE` packets have fields indicating the source and destination key-value store migration instance, filled with the server IP address and transport layer port pair `(SRC_IP, SRC_Port, DST_IP, DST_Port)`. `MIGRATE_GROUP_START` and `MIGRATE_GROUP_COMPLETE` packets notify switch that a migration group from DBPort instance has started migration or has completed migration, to update migration status tracking indexing in the switch. `MIGRATE_DATA` packets simply carry the key-value pairs in

the packets and are transferred from the source server to the destination server.

Network Routing. NetMigrate leverages existing routing protocols to forward packets. For migration control and migration data packets, they are routed as normal packets from the source server to the destination server, in addition to updating indexing data structure in the switch. NetMigrate switches are placed on the path from the clients to the storage clusters. Client query and reply packets are forwarded based on indexing look-up results to determine the "right" storage server as described in § 4.3.

Merging read replies with version control. NetMigrate has an 8-bit version control field in reply packets, identifying: (1) whether the query is successfully executed in the backend storage server, (2) the reply packet is from the source server or the destination server, (3) whether the reply is from double-read, and (4) whether the query needs PriorityPull. Two more bits are reserved for more controls. To handle double-reads, the client agent merges two replies received from the source and the destination to the one with a newer version.

B Artifact Appendix

Abstract

NetMigrate is a key-value store live migration protocol by leveraging programmalbe switches. NetMigrate migrates KVS shards between nodes with zero service interruption and minimal performance impact. During migration, the switch data plane monitors the migration process in a fine-grained manner and directs client queries to the right server in real time.

Our artifact provides code and scripts to reproduce experimental results in the paper, especially in replicating Figures 4-7. We demonstrated experimental results on three commodity machines and a Barefoot Tofino switch.

Scope

The artifact can be used as a prototype of NetMigrate migration protocol with the backend KVS as Redis, and to validate the experimental results on performance improvement compared with other migration baselines in the paper.

Contents

The artifact contains four migration protocols' server agents in `cpp/server` folder, YCSB client implementation for four migration protocols in `cpp/YCSB-client`, and switch data-plane and control-plane code for NetMigrate in `tna_kv_migration`, with experiment steps in `README.md` and `experiment_steps` folder.

Hosting

The artifact is hosted on GitHub (https://github.com/Froot-NetSys/NetMigrate, main branch, commit c977bfa2c8eeec7d77fb4a834cebfb1e3f819e24).

Requirements

We developed and tested the artifact on the below platform:

- **Hardware:** A Barefoot Tofino switch, and three servers each with a NIC (we used an Intel XL710 for 40GbE QSFP+) and multi-core CPU, connected by the Tofino switch.
- **Software:** Tofino SDK (version 9.4.0) on the switch, Python2.7 on the switch, and gRPC 1.50.0 and protobuf 3.21.6.0 for PriorityPulls in KV servers.

IONIA: High-Performance Replication for Modern Disk-based KV Stores

Yi Xu[‡ §]
University of California, Berkeley

Henry Zhu[‡]
University of Illinois Urbana-Champaign

Prashant Pandey
University of Utah

Alex Conway
Cornell Tech and VMware Research

Rob Johnson
VMware Research

Aishwarya Ganesan
University of Illinois Urbana-Champaign and VMware Research

Ramnatthan Alagappan
University of Illinois Urbana-Champaign and VMware Research

Abstract. We introduce IONIA, a novel replication protocol tailored for modern SSD-based write-optimized key-value (WO-KV) stores. Unlike existing replication approaches, IONIA carefully exploits the unique characteristics of SSD-based WO-KV stores. First, it exploits their interface characteristics to defer parallel execution to the background, enabling high-throughput yet one round trip (RTT) writes. IONIA also exploits SSD-based KV-stores' performance characteristics to scalably read at any replica without enforcing writes to all replicas, thus providing scalability without compromising write availability; further, it does so while completing most reads in 1RTT. IONIA is the first protocol to achieve these properties, and it does so through its storage-aware design. We evaluate IONIA extensively to show that it achieves the above properties under a variety of workloads.

1 Introduction

Key-value stores play a central role in datacenter applications. Today, many KV stores such as LevelDB [31], RocksDB [26], Cassandra [2] and others [8, 15, 62, 64] are built using *write-optimized indexes* (WOIs) such as LSMs [58]. We refer to these stores as WO-KV stores. WO-KV stores have become a popular choice because they offer significantly higher write performance than B-tree stores [9, 62]. Further, recent WO-KV stores are optimized for modern SSDs, and can extract their high bandwidth and offer low latencies [73].

As KV stores are increasingly deployed in datacenters, making them fault tolerant is critical. A common way to achieve this goal today is to replicate the store on many machines and use off-the-shelf replication protocols like MultiPaxos [43], Raft [56], or Viewstamped Replication (VR) [46] to coordinate writes and reads to the store. For example, ZippyDB at Meta [65, 68] uses MultiPaxos to replicate RocksDB. Several other systems use a similar layered design [11, 20, 21, 47].

Unfortunately, however, using off-the-shelf protocols to replicate modern WO-KV stores squanders their high write performance (§2). These protocols offer low write throughput because they must apply writes sequentially on a single thread to ensure that the replicas are identical [66]. They also incur high latencies because replicas must coordinate to agree on the order of writes. Although many prior protocols [13, 37, 42] have been proposed to safely apply writes on multiple threads for high throughput, they incur high latencies. At the same time, many prior approaches [28, 44, 59, 61] that achieve low-latency writes suffer from low throughput. Existing replication approaches thus cannot preserve the high write performance of WO-KV stores.

Off-the-shelf protocols also lead to poor read performance. For strong consistency, these protocols restrict reads to a designated replica called the leader [38, 52, 55]. Thus, the read bandwidth of the followers goes unused. This is particularly bad for WO-KV stores because of their read-write asymmetry [9]: reads are more expensive than writes and thus performance drops with more reads. As a result, serving all reads at the leader pushes it to a less performant regime, impairing overall throughput. Many prior protocols have devised ways to scalably read from followers. However, as we discuss (§2), many of them suffer from high latencies [11, 74]; others that offer low latencies do so by (regrettably) trading off write availability and slowness tolerance [14, 27, 39, 70].

An ideal protocol must preserve the high write performance of WO-KV stores, offering high throughput and low latency. It must also safely (i.e., with consistency) scale reads while offering low-latency reads without impacting availability. We observe that a main reason why existing approaches do not achieve these ideal properties is that they are largely *oblivious* of the underlying SSD-based WO-KV store's characteristics.

In this paper, we design IONIA, a novel replication protocol that carefully exploits the interface and performance characteristics of the underlying SSD-based WO-KV store. We show that such careful storage-aware design enables IONIA to achieve high-throughput, 1RTT writes, and scalable, 1RTT reads without impacting availability (§3).

IONIA avoids the high latency of writes inherent in prior parallel-execution protocols [13, 37, 42] by exploiting the interface atrributes of WO-KV stores, improving the write path. In particular, WO-KV stores convert all writes into *blind* writes to avoid performing a slow read before a write. Consequently, they do not return an execution result but only an acknowledgment to clients when writes complete; i.e., the update interfaces in WO-KV stores are nil-externalizing, a

[‡]Co-primary authors
[§]Did part of the work during an internship at VMware Research

property we identify in our prior work on Skyros [28]. Thus, similar to Skyros, IONIA needs to only guarantee durability when writes complete, which can be achieved in 1RTT without coordination. IONIA then defers ordering and parallel execution to the background, achieving both high-throughput and 1RTT writes.

IONIA's main novelty is its scalable, 1RTT read path. We first observe that existing low-latency, scalable read protocols conflate scalability and locality due to their focus on in-memory stores. Locality means that reads can be locally served by any replica without additional messages to other replicas. This conflation, however, compromises write availability and slowness tolerance: because clients must be able to locally read at any replica, writes are replicated to *all* replicas (not just a quorum). Locality is a necessary condition for scalability for in-memory stores, where network messages are the bottleneck. In contrast, we realize that for SSD-based stores, *scalability can be decoupled from locality*: because SSD is the bottleneck (instead of network), even non-local reads that send additional messages won't impact scalability as long as these additional messages access only in-memory state (not the SSD) on other replicas.

Based on this insight, IONIA replicates only to a quorum for availability while allowing reads from any replica. Then, to handle lagging followers, for every follower read, IONIA performs a check (which we call a meta query) at the leader to validate the result returned by the follower. However, this approach can still scale because the leader serves meta queries from memory at high throughput. Overall, compared to prior protocols that focus on in-memory stores, IONIA exploits the performance gap between SSDs and DRAM to decouple scalability from locality, achieving scalable reads.

Second, while the meta-query approach offers scalability, it in itself does not offer 1RTT reads, a requirement for modern SSD-based stores that offer low latencies. To achieve 1RTT reads, IONIA sends the read to a follower and the meta query to the leader in parallel. However, a challenge is that, since the requests are concurrent, the leader cannot directly indicate to the client whether or not the follower's result is up-to-date. To solve this problem, IONIA proposes a *client-side consistency-check* mechanism, where the leader returns enough information about the key being read and the client makes the decision about the freshness of the follower's result.

We have designed and implemented IONIA (§4). A main challenge in our design is to ensure that meta queries can always be served from the leader's memory. IONIA addresses this by maintaining a compact history of recently modified keys instead of all keys in the store. A related challenge is how to ensure that the leader returns the correct information for meta queries for keys not present in the history. IONIA solves this problem by returning slightly inaccurate information for such keys but without endangering consistency. We have model checked IONIA to show its correctness.

Our evaluation (§5) shows that for writes, IONIA matches the throughput of parallel-execution protocols while offering the low-latency of Skyros; IONIA also approximates the performance of an unreplicated server. For reads, IONIA offers linear scaling, saturating the read bandwidth of all replicas without meta queries becoming the bottleneck. With mixed workloads, IONIA offers 1.8× higher throughput than IONIA-LR (a variant where reads are restricted to leader). We show that most reads finish in 1RTT and that this is achieved with small histories (e.g., 50MB). With YCSB [19], IONIA improves throughput by 16× to 38× over MultiPaxos.

This paper makes three contributions.

- We first show how designing a replication protocol by paying attention to the underlying SSD-based WO-KV store layer yields desirable properties.
- Second, we present novel ideas such as decoupling scalability from locality and client-side consistency checks, which enable IONIA to achieve scalable and low-latency reads without compromising availability or slowness tolerance.
- Finally, we present a thorough experimental evaluation, showing IONIA's benefits.

2 Background and Motivation

We provide background on WO-KV stores and building replicated KV stores. We discuss why existing protocols are insufficient for WO-KV stores, leading to undesirable properties.

2.1 WO-KV Stores Background

KV stores are implemented using a disk-based index structure. Stores built using B-tree or its variants [57] are a poor fit for write-intensive workloads because writes in B-trees require random IO, which is significantly slower than sequential IO. As a result, modern KV stores have turned towards write-optimized indexes (WOI) such as LSMs [58] or B^{ε}-trees [12]. WOIs offer higher write throughput than B-trees because they batch writes and sequentially transfer large batches to disk, amortizing IO cost [9]. Today, many local KV stores including LevelDB [31], RocksDB [26], and several others [60, 62, 64, 73] are built atop WOIs. WO-KV stores are also used as storage engines in distributed systems like BigTable [15], Cassandra [2], and CockroachDB [17].

Read-Write Asymmetry. In B-trees, both writes and reads require random IO and thus both are limited by device's random IOPS [9]. WOIs provide the same read performance as B-trees [9] and are limited by random IOPS. However, writes in WOIs are limited by sequential bandwidth. Consequently, WOIs have a *read-write asymmetry* in performance: writes in WOIs are much faster than reads.

An important implication of this asymmetry is that KV stores built using WOIs avoid *query-before-update* [9]; issuing a query before an update squanders the benefits of WOIs, essentially making WOIs behave like B-trees. As a result, WO-KV stores convert all writes into blind writes. For example, a *put* simply absorbs a write to a key without checking if the key is already present since the check would

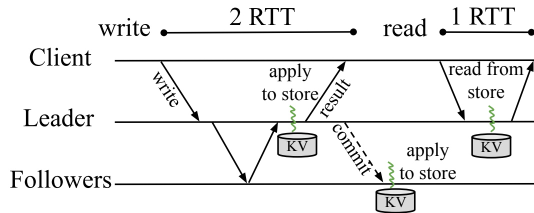


Figure 1: **Off-the-Shelf Replication.** *The figure shows how writes and reads are processed in replicated stores built using off-the-shelf protocols.*

require a (slow) read [50]. Similarly, a *delete* blindly inserts a tombstone without checking for key presence [67]. Even read-modify-writes (RMW) are transformed into blind upserts [9]; as an example, RocksDB implements upserts using merge [25]. An upsert encodes a RMW by specifying a key k and a function f that changes the value of k. k and f are then blindly recorded; the value is evaluated later only when needed (e.g., on a read). This, in turn, means that when writes complete, WO-KV stores do not reveal system state to the clients [28]: they do not return an execution result or execution error (e.g., to indicate key presence or absence) but only an acknowledgment. Our prior work calls such updates nil-externalizing or nilext [28]. Note that while a nilext interface does not return *execution* errors, it can still return validation errors (e.g., malformed requests, unreachable server).

2.2 Consistent Replicated KV Stores Background

A common way to build a strongly consistent replicated KV store is to layer an off-the-shelf replication protocol like MultiPaxos atop the local KV store. Many systems [11, 20, 21, 47] including Meta's ZippyDB [65, 68] use such a layered design.

Figure 1 shows how writes and reads are processed in a replicated KV store built using off-the-shelf replication protocols such as MultiPaxos (or Raft [56] or VR [46]). As shown, clients submit writes to the leader, which orders the requests by appending them to its consensus log; the leader then sends the request to the followers. The followers append to their logs and respond. Once a majority has agreed to the order, the leader applies the writes to the KV store and returns the result. Asynchronously, the leader sends a *commit*, upon which all followers apply the ordered writes. All replicas apply writes sequentially on a single thread to avoid non-determinism. Applying writes on multiple threads is non-deterministic [42]: replicas may apply writes in different orders, causing the KV store state across replicas to diverge.

Because followers could be lagging and thus could return stale data, off-the-shelf protocols allow reads only at the leader to ensure strong consistency [38, 48, 52, 55]. To prevent a deposed old leader from serving stale data, these protocols employ leader leases [38, 46], which ensures that a new leader is elected only after the old leader's lease has expired.

2.3 Why Are Existing Protocols Insufficient?

We first explain the drawbacks of off-the-shelf protocols to replicate SSD-based WO-KV stores. Prior work has built

	Protocol/System	Write throughput	Write latency	Scalable reads	Read latency	HA writes	Slowness tolerance
	Unreplicated	high	1RTT	×	1RTT	×	×
	MultiPaxos [43], Raft [56], VR [46]	low	2RTT	×	1RTT	√	√
Improving Writes	Parallel execution (e.g., CBASE [42], Eve [37])	high	2RTT+Δ	×	1RTT	√	√
	Specpaxos [61], CURP [59], Skyros [28]	low	1RTT	×	1RTT	√	√
Improving Reads	Gaios [11]	low	2RTT	√	2.5RTT + Δ	√	√
	CRAQ [70]	high	(n/2)RTT*	√	1RTT	×	×
	Hermes [39]	high	2RTT	√	1RTT	×	×
	Ionia	high	1RTT	√	1RTT	√	√

Table 1: **Existing Approaches.** *The table compares different existing approaches. Δ: waiting delay for bigger batches (in parallel execution) or followers to catch up (in Gaios); *: CRAQ builds upon chain replication [71] which incurs O(n) RTTs for writes; n: number of replicas.*

optimized protocols to improve over off-the-shelf replication. However, as we discuss, these protocols are still not the ideal choice for SSD-based WO-KV stores, leaving a huge room for improvement in performance and availability.

We will use an unreplicated server as a baseline to show the drawbacks of existing protocols. As shown in Table 1 (first row), an unreplicated server offers low latency for writes and reads because clients can submit a request to the server and get a response in 1RTT. The unreplicated server can also offer high throughput because it can safely apply writes on multiple threads. However, the unreplicated server has an obvious problem: it cannot tolerate server failure or slowness. Further, read performance is limited by the single server.

An *ideal* system must tolerate failures and slowness while matching the unreplicated server's latency and write throughput. It must also scale read performance with replicas.

2.3.1 Off-the-Shelf Protocols are Ill-Suited for WO-KV

Off-the-shelf protocols like MultiPaxos tolerate failures and slowness (Table 1 second row). However, these protocols offer significantly lower throughput than an unreplicated server because they must apply writes sequentially on a single thread to avoid non-determinism. Modern WO-KV stores, however, are optimized for multi-core CPUs and SSDs [18, 26], where ingesting data using many threads is necessary for high throughput. These protocols also incur high latencies for writes because the replicas must coordinate to order the writes in the critical path. Specifically, writes incur 2RTTs (client→leader→followers→leader→client), doubling the latency of an unreplicated server (client→server→client). Modern WO-KV stores offer low latencies [18] and thus latency from additional RTTs is undesirable. As we soon show, such eager coordination is, in fact, unnecessary for WO-KV stores.

Off-the-shelf protocols restrict reads to the leader. Because the leader sees all writes, with leader leases [46], reads are guaranteed to see the latest data, completing them in 1RTT.

However, this means that reads cannot scale with replicas. Restricting reads to the leader also reduces *overall* throughput because reads contend with writes. This is particularly bad for WO-KV stores because of read-write asymmetry, where performance drops with more reads. If the leader processes both reads and writes, the ratio of reads to writes will be higher than a system that splits the read load across replicas. Since WO-KV stores offer lower performance with more reads, the leader will operate at a lower throughput regime than a system that spreads reads. Since the leader's throughput is the bottleneck, the overall performance suffers.

2.3.2 Shortcomings of Existing Improvements

We now discuss the vast body of work that has attempted to improve the write and read performance of off-the-shelf protocols. We discuss their shortcomings and argue why they are not a great fit for replicating WO-KV stores.

High-Throughput *Or* Low-Latency Writes. As we discussed, applying writes on multiple threads is essential to realizing high throughput in modern WO-KV stores. Fortunately, prior protocols such as CBASE [42], Eve [37], and others [1, 13, 24] (third row) have shown how to leverage multi-threaded execution in replicated systems while avoiding inconsistencies. For correctness, these protocols concurrently execute only non-conflicting writes; conflicting writes are serialized and applied in the same order across all replicas. Thus, these protocols can offer high write throughput and could serve as a good base for replicating WO-KV stores. Unfortunately, however, these protocols incur high latencies similar to off-the-shelf protocols. In fact, the latency can be higher than off-the-shelf protocols because these protocols must create bigger batches to find opportunities to concurrently apply many writes. While prior protocols have proposed techniques for low-latency writes by exploiting commutativity [44, 53, 59], speculation [41, 61], network ordering [45], and interface semantics [28] (fourth row), these protocols apply writes sequentially and thus suffer from low throughput. Existing protocols to improve write performance thus either suffer from high latencies or low throughput.

Scalable Reads: Write Availability *Or* Low Latency. Prior work has proposed ways to scale reads by allowing reads at all replicas. The main challenge these protocols must solve is to ensure that a read from a replica returns the most up-to-date data; a stale read will violate strong consistency [33].

Systems like Gaios [11] and Gnothi [74] (fifth row) ensure consistency by routing all reads to the leader; the leader then dispatches the read to a follower. The follower next waits until it has caught up with the leader before serving the read. While this approach scales reads, it incurs several RTTs and waiting delay. This was acceptable for spinning disks where disk latency was much higher than RTTs. However, for modern SSD-based WO-KV stores, incurring multiple RTTs increases latency considerably compared to an unreplicated server.

A few protocols [14, 27, 30, 39, 70] achieve scalability while offering 1RTT reads (sixth row). These protocols target in-memory stores and thus, network IO is often the scalability bottleneck. Therefore, for scalable 1RTT reads, these protocols ensure a read at a replica is always *local*: a client locally reads at a target replica without additional messages to other replicas. However, to enable local reads, these protocols must replicate writes to *all* replicas (not just a quorum) before writes can complete. This conflation of scalability and locality, however, has a critical impact on availability because writes cannot complete if any replica fails [30][†]. Further, they cannot exclude slow replicas when processing writes. We thus observe a *fundamental tradeoff* in these protocols: they offer scalable reads with low latency but do so at the expense of availability and slowness tolerance.

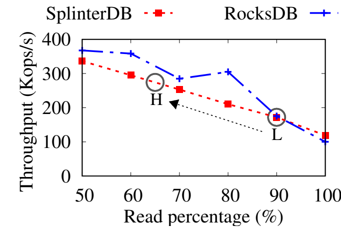


Figure 2: **Read-Write Asymmetry.** *The figure shows how performance decreases with more reads in WO-KV stores.*

Summary. Off-the-shelf protocols suffer from poor performance and thus are not suitable for building replicated WO-KV stores. While many solutions have been proposed to improve writes, they do not achieve high throughput and low latency simultaneously. A few solutions have been proposed to achieve scalable reads. These protocols either compromise on latency, or tradeoff write availability and slowness tolerance for low latency. Further, protocols that improve writes suffer from poor read performance (e.g., parallel-execution protocols do not scale reads); similarly, scalable read protocols do not offer optimal write performance (e.g., CRAQ incurs O(n) and Hermes incurs 2RTTs for writes).

We next show how a protocol that carefully exploits the characteristics of the underlying SSD-based WO-KV store can advance beyond existing approaches and offer high-throughput 1RTT writes, and scalable 1RTT reads without compromising on availability or slowness tolerance (last row).

3 IONIA Ideas and Protocol Overview

We now describe the key insights behind IONIA and provide an overview. The next section presents the detailed design.

3.1 Key Insights and Ideas

Deferred Parallel Execution with Immediate Durability. IONIA achieves high-throughput writes with low latency. To achieve high throughput, IONIA employs techniques from prior parallel-execution protocols and concurrently executes

[†]CRAQ and Hermes must wait for an expensive reconfiguration to complete before the system can become available after failures.

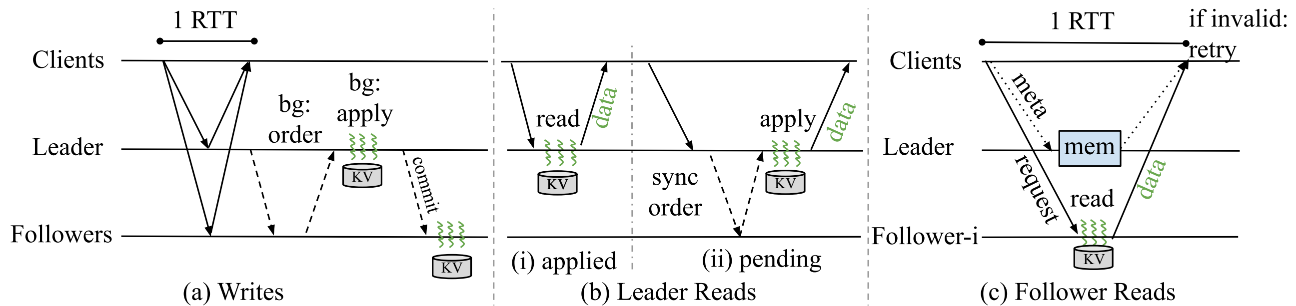


Figure 3: **Ionia Protocol Overview.** *The figure shows how* Ionia *handles writes, leader reads, and follower reads.*

only non-conflicting writes to avoid inconsistencies. However, existing high-throughput protocols incur high latencies. We realize this latency cost can be avoided by exploiting the interface semantics of WO-KV stores. Because WO-KV stores convert all writes into blind writes, they do not return an execution result; i.e., the writes are nilext. Thus, similar to Skyros [28], Ionia need not immediately order and apply writes to the underlying WO-KV store. Instead, it must only guarantee that writes are durable before acknowledgment. Ionia achieves durability in 1RTT by having the clients write to many replicas in parallel without coordination across replicas. Ordering and parallel execution of writes are deferred to the background. Thus, Ionia combines the benefits of prior parallel-execution and low-latency approaches to achieve both high-throughput and 1RTT writes.

Decoupling Scalability from Locality. Existing low-latency, scalable read protocols conflate scalability and locality as they focus on in-memory stores. To achieve scalability, they require that reads are local (no additional messages to other replicas). Locality is necessary for scalability in in-memory stores where network messages are the bottleneck. This conflation, however, impacts availability and slowness tolerance. We realize that in SSD-based WO-KV stores, SSD random IOPS is the bottleneck, not the network messages. Thus, reads can scale even if they are non-local (i.e., they send additional messages to other replicas), as long as the additional messages *access only in-memory state (not the SSD)* on other replicas. That is, scalability can be *decoupled* from locality in SSD-based stores.

Based on this insight, Ionia writes only to a quorum for availability and slowness tolerance and allows reads at any replica. The challenge, however, is that a read at a replica cannot rely on just the local state because the replica might be lagging. Ionia addresses this challenge as follows. For every read at a follower, Ionia sends a meta query to the leader (which is guaranteed to have seen all writes). The meta query helps identify whether or not the result returned by the follower is up-to-date.

A critical requirement for scalability is that meta queries must be cheap; in particular, meta queries must avoid SSD IO. Otherwise, the meta-query throughput could saturate before the replicas' collective SSD IOPS are saturated, limiting scalability. We soon discuss how Ionia can always serve meta queries from memory without SSD IO (§4). Two factors help ensure that in-memory meta queries will not become the bottleneck in practice. First, reads in WO-KV stores are limited by random IOPS, which is ∼600K IOPS [35] in today's fast SSDs. On such hardware, even with caching and skewed workloads, modern WO-KV stores offer only about 850K reads/s [18]. Second, most systems typically use a small replication factor (3 or 5) [34]. However, even an unoptimized server in our setup could handle 12M meta-query-like RPCs/s, which is well over the collective read bandwidth ($5 * 850$K). Thus, meta queries will not be the scalability bottleneck.

By spreading the read load, Ionia also improves overall performance under mixed workloads. In WO-KV stores, as reads increase, the performance drops due to read-write asymmetry as illustrated in Figure 2. Thus, under mixed workloads, taking off reads from the leader and spreading it across replicas, reduces its read-write ratio which improves leader's throughput. For example, consider a workload with 90% reads and 10% writes; this is a low performance regime for WO-KVs (L in Figure 2). If the 90% reads are split across five replicas, the leader would process 18% reads and 10% writes (i.e., read-write ratio is 65:35), which is a higher performance regime (H). Overall, distributing reads improves the leader's throughput; since the leader's storage layer is the bottleneck, improving it boosts the overall performance.

Low-latency Reads via Client-side Consistency Checks. While the meta-query approach enables scalability, it in itself does not ensure 1RTT read, which is important for modern SSD-based stores that offer low latencies. To achieve 1RTT and scalable reads, our main idea is to have clients send the meta query to the leader and the actual read to a follower *in parallel*. However, this raises a challenge – the leader cannot definitively determine whether or not the data returned by the follower is up-to-date. This is because the leader does not know the follower's status at the time the read was performed at the follower. To solve this, Ionia uses a novel *client-side consistency check*, where the leader returns information about the key being read, and the client makes the final decision about the freshness of the follower's data. If the client determines that the data is stale, it retries the read. Client-side checks offer 1RTT reads while ensuring strong consistency.

3.2 IONIA Protocol Overview

We now provide an overview of IONIA. Figure 3(a) shows the write path. Clients send writes to all replicas in parallel. The replicas make the writes durable without any coordination and respond directly to clients. Once a client receives enough replies including one from the leader, the write is complete; clients can proceed without waiting for the writes to be ordered or applied. Thus, all writes complete in 1RTT. The 1-RTT write path is similar to Skyros [28].

While IONIA does not order and apply writes in the critical path, writes must be eventually ordered and applied in the real-time order to preserve linearizability [33]. That is, if operation *y* starts after another operation *x* completes, then *y* must be ordered after *x*. However, when concurrently executing requests, non-conflicting writes (or, writes that update different keys) can be applied in any order across replicas. Only conflicting writes that update the same key must be applied in the same real-time order across replicas. IONIA ensures such correct write ordering in the background. The leader periodically determines the order for the writes and gets enough followers to accept the order; all replicas then apply conflicting writes in the order determined by the leader in the background.

We next discuss the read path (see Figure 3(b) and (c)). Reads can be served by any replica. When a read to a key arrives at the leader, there might be updates to the key that have not been ordered and executed yet. This is because IONIA orders and executes writes only lazily. Thus, in IONIA, before the leader can serve a read, it must first check if there are pending updates to the key being read. If no, the leader serves the read immediately (3(b)(i)). If there are pending updates, then the leader synchronously orders and executes the updates before serving the read (3(b)(ii)). Fortunately, such slow-path reads are rare in practice due to two reasons. First, the leader keeps ordering and executing writes in the background; thus, in most cases, updates are executed already by the time a read arrives. Second, traces from deployed systems [23] show that reads to recently written objects are rare [28].

In IONIA, reads are served by followers as well. To prevent clients from seeing stale data from lagging followers, in addition to sending the actual read request to a follower, clients also send a meta query to the leader as shown in 3(c). The follower locally reads its KV store and returns the kv pair. The leader responds to the meta query with information about the key being read (in particular, the latest update applied to the key). IONIA ensures that the leader can get this information from memory without an SSD IO. The client uses this information to decide whether or not the result from the follower is valid. If valid, the read completes in 1RTT; in contrast, if the data is stale, the client retries the read at the leader.

4 IONIA Design and Implementation

We use viewstamped replication (VR) as a baseline to describe IONIA's design. VR is leader-based and makes progress through a sequence of views. In each view, one replica serves as the leader. VR tolerates *f* failures with *2f+1* replicas and offers linearizability. Upon writes, the leader appends requests to a *consensus log* and sends a *prepare* to the followers. Once *f* followers acknowledge via a *prepare-ok* (after adding to their logs), the leader applies the writes and returns the result to the client. Reads are served by the leader and the system uses leader leases for consistency. The replication layer interacts with the KV store via upcalls; writes are applied via the *Apply* upcall and reads are served via the *Read* upcall.

```
MakeDurable(w) // add write w to durability log
AddToExecQueuesWithDeps(B)
// add batch to execution queues with dependencies
Apply(w) // apply write w to KV store
LeaderRead(k) // returns data, must_sync
MetaQuery(k)
// returns must_sync, modified_index for k
FollowerRead(k) // returns data, applied_index
```

Figure 4: **Storage-system Upcalls in IONIA.**

IONIA is also leader-based and offers the same guarantees as VR. IONIA augments the interface between the replication layer and the storage system with additional upcalls as shown in Figure 4. These upcalls help IONIA handle different operations. We first explain how IONIA handles writes (§4.1) and reads (§4.2) during normal operation. We then describe how IONIA handles failures and view changes (§4.3). Finally, we present correctness proof sketch (§4.4), model checking results (§4.5), and implementation details (§4.6).

4.1 Writes

IONIA's goal is to achieve high-throughput writes with low latency (1RTT). Our idea to achieve this end is to apply writes on multiple threads but avoid high latency by deferring ordering and execution of writes to the background. IONIA can defer ordering and execution because WO-KV stores do not return an execution result. IONIA must only ensure that writes are durable before acknowledgment (i.e., they will not be lost even if f replicas fail).

To achieve low-latency durability, IONIA borrows the idea of durability logs from Skyros [28]. IONIA clients directly send writes to all replicas. Each replica adds the write to a separate durability log (via the *MakeDurable* upcall); the replicas then respond directly to clients without any coordination, completing writes in 1RTT. Intuitively, the durability logs contain writes that are not yet ordered and applied. A client waits for a supermajority ($f + \lceil f/2 \rceil + 1$) acknowledgments including one from the leader to complete a write. Because the leader's response is required to complete writes, the leader's durability log captures the correct (real-time) ordering. The leader uses this property to order requests in the background during normal operation. When the current leader fails and a new one is elected, supermajority quorums enable IONIA to reconstruct the linearizable order as we soon discuss (§4.3).

Although the leader's durability log captures the correct

order, the order is *not finalized* yet as writes might be present in different orders in durability logs across replicas. This is because writes are added to durability logs without any coordination. Thus, the leader must finalize the order before the writes can be applied to the KV store. To do so, the leader periodically gets a majority of replicas (including self) to agree on the order. The leader adds requests from its durability log to its consensus log in order and sends a *prepare*. The followers append the requests to their consensus logs and respond with *prepare-ok*. Once *f* followers respond, the ordering is finalized. The leader can now apply the writes to the store. The leader also sends a *commit* which informs the followers of the latest consensus-log index up to which the order has been established; the followers can apply writes up to that index. To improve throughput, the leader batches several requests in a single *prepare*; however, this batching does not affect client-visible latency since the ordering (and execution) happen entirely in the background.

Once ordering is established, each replica applies writes to the store in multiple threads. The replicas execute only non-conflicting writes in parallel. Conflicting writes are executed serially and in the order they appear in the consensus log; this ensures that conflicting writes are executed in the same order across replicas. IONIA realizes the above idea as follows. Each replica maintains a set of execution queues, one for each execution thread; an execution thread applies writes from its queue in order. However, before applying a write *w*, IONIA must ensure that all writes that conflict with *w* and appear before *w* in the consensus log have been executed. To capture and set these dependencies, the replication layer invokes *AddToExecQueuesWithDeps* when a batch of writes has been ordered. The storage layer captures conflicting writes to the same key by adding writes to a particular key to the same queue in the order they appear in the batch; the replicas use a deterministic hash of the key to achieve this. Other conflicts between requests can be captured by explicitly annotating a request *w* with requests from other queues that must be executed before *w*. For example, a multi-key write can be added to any queue with explicit dependencies to the other requests in the queues that conflicts with *w*.

To execute requests, each execution thread retrieves a request *w* from its queue and waits until the dependencies of *w* are executed before it executes *w*. Once a replica has applied a request to the KV store, it removes it from its durability log. The replica also updates its *applied-index*, the latest index in the consensus log up to which it has applied to the KV store. The leader keeps learning each follower's applied-index. Figure 5 shows how writes are ordered, assigned to execution queues, and finally applied to the KV store.

4.2 Reads

IONIA's goal is to provide scalable, 1RTT reads without impacting availability. As we discussed earlier, to ensure strongly-consistent reads, IONIA issues a meta query to the leader to check the validity of a read at a follower. IONIA ensures scalability by serving the meta queries from the leader's memory while the reads at the replicas are bound by SSDs' random IOPS. Further, to achieve 1RTT reads, IONIA clients send the actual read request and meta query in parallel and employ a client-side check to validate the returned data. We now explain how the meta query and client-side consistency check mechanisms work. We then describe how IONIA ensures that the leader can always serve the meta query from its memory by maintaining a compact history.

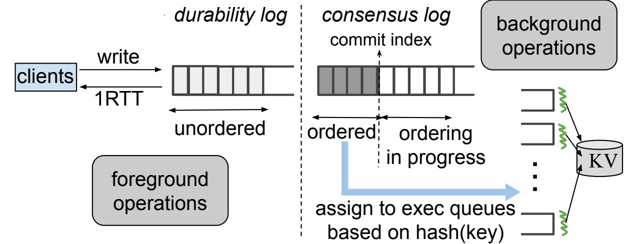


Figure 5: **IONIA Write Processing at a Replica.** *The figure shows how writes are processed in one replica (other replicas are not shown).*

4.2.1 Meta Queries and Client-side Consistency Check

Reads at the leader can be served directly without any cross-replica checks because the leader is guaranteed to have seen all writes. However, when a read arrives at the leader, there might be unordered (and hence not-yet-applied writes) in its durability log. Thus, upon a read, the leader first checks if there are such pending updates to the key being read (via the *LeaderRead* upcall). If there are no pending updates, the leader reads and returns the kv pair, finishing the read in 1 RTT. If there are pending updates, the upcall returns a *must_sync* flag. The replication layer then synchronously appends the pending writes from the durability log to its consensus log, gets the followers to agree, applies the writes to the KV store, and finally reads and returns the kv pair. In this case, the read completes in 2 RTTs. However, such synchronous reads are rare as we discussed (§3) and will show in our evaluation.

Reads at the followers require a check at the leader because followers could be lagging. To read at a follower, a client sends the read to the follower and a meta query to the leader. To enable 1RTT reads, both requests are sent in parallel. The meta query specifies the key being read. Upon receiving a read, the follower reads the kv pair from the store (via the *FollowerRead* upcall) and returns it. To answer meta queries, the leader maintains a *history*; this history maps a key to the consensus-log index corresponding to the latest write that modified the key. Upon a meta query, the leader first checks its durability log to see if there are pending updates to the key *k* being read. If there are none, the leader queries the history and obtains the latest index *i* that modified *k*.

One way to implement the meta queries is to have the leader make the decision about freshness the of follower's data. Specifically, the leader can compare the follower's applied-

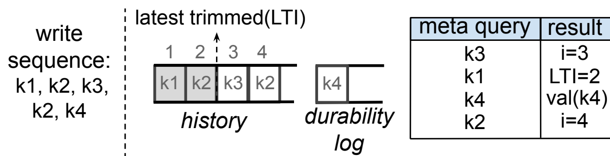


Figure 6: **History and Meta Queries at Leader.** *Writes completed: k1, k2, k3, k2, k4; history trimmed up to 2, so LTI=2. Meta queries for k3 and k2 return the actual modified index; query for k1 returns LTI because k1 is not in history; query for k4 returns value of k4 after synchronous execution.*

index a_f (which it learns periodically) and i. If $a_f > i$, then the leader could decide that the follower is to up-to-date. But this approach is *incorrect*. This is because the follower could return an older value, then apply a later update, and then update a_f at the leader. Now, if the meta query reaches the leader, it would incorrectly determine that the follower's result is valid, violating strong consistency.

To address this issue, IONIA pushes the check to the client. In addition to the kv pair, the follower also returns its own *applied-index*. Note that the follower reads its applied-index a *before* reading from the KV store, so that the kv pair returned is never older than a, even with concurrent writers. The leader, instead of making a decision locally, returns the latest index i that modified k. The client then performs the check by comparing a and i. If $a >= i$, then the follower has applied all the updates to k and thus the returned data is up-to-date, finishing the read in 1RTT. If $a < i$, then the follower has not applied all updates to k and thus the data is stale. In such cases, the check fails and the client retries the read at the leader.

When there are pending updates in the durability log for key being read, the leader knows that the follower's data will be stale and could instruct the client to retry. However, this would increase latency. IONIA optimizes this case by having the leader synchronously order and execute updates and return the actual read result. The client then ignores the result from the follower and uses the one from the leader.

4.2.2 Cheap Meta Queries with Compact History

The history at the leader logically needs to maintain the latest consensus-log index that modified every key. To ensure meta queries are fast, the history must be maintained in memory, not disk. However, maintaining the modified index for *every* key in memory is impractical for large disk-based stores.

To solve this problem, IONIA maintains the history only for recently modified keys. A key is added to the history when a write to it is recorded on the consensus log. The leader periodically learns each follower's *applied-index*. When all followers have applied upto i, the history upto i is trimmed. This raises a problem, however: when a meta query arrives for a key k, the history may not contain k. Note that a key k being absent from the history means that all followers have applied all writes to k. However, the leader still *cannot* indicate to the client that the data from the follower is up-to-date. This is because, the leader doesn't know which version the follower returned. Specifically, the follower could have returned an older version and then applied the latest update, causing the leader to trim the history before the meta query arrives.

What must the leader return when k is not in the history? Intuitively, the leader must return an index greater than or equal to the actual modified index. Returning anything smaller is unsafe: the client may incorrectly believe the follower has given the latest data. Thus, when the leader does not find a key in its history, it returns the index that was last trimmed from the history; we call this the last-trimmed index or *lti*. Returning *lti* is correct, because if the modified index was part of the trimmed history, *lti* would be greater than or at least equal to the modified index. Figure 6 shows how the leader maintains the history and returns results for meta queries.

Optimizations. IONIA uses two techniques to optimize the procedure described so far. First, it uses lazy history trimming. The history could be trimmed up to i *immediately* after the leader learns that all followers have applied up to i. However, such eager trimming can lead to inefficiencies. For example, consider a case where a client reads at a follower and get an *applied-index* a. After this, the follower applies m more writes and informs the leader of its *applied-index* $a+m$; the leader trims history upto $a+m$. If the meta query now arrives at the leader, the history would not contain the key. The leader would return $a+m$ (its lti), causing the client check to fail. To avoid such scenarios, IONIA only lazily trims the history.

Second, a follower that has been disconnected for long or failed could prevent the leader from trimming the history. To avoid this problem, the leader maintains a set called active-followers. Failed followers are removed from the set. The leader waits only for the followers in the active-followers set to trim the history. Thus, reads at a disconnected follower that has missed writes will always be rejected because its *applied-index* will be less than the index returned by the leader.

Summary. Figure 7 summarizes IONIA's read protocol. To read key k at the leader, the clients invoke LEADER_READ. If there are no pending updates in the durability log, the leader reads and returns k. If there are pending updates to k, the leader orders and applies them, after which it reads and returns k. When reading at a follower, the client parallely invokes FOLLOWER_READ and META_QUERY. The follower reads k and returns it along with its *applied-index*. In META_QUERY, the leader first checks if there are pending updates to k. If yes, the leader orders and executes the pending updates, and returns the actual data and indicates this in a flag. If there are no pending updates, the leader returns the modified index (if the key is in the history) or the last-trimmed index. The client finally compares the results and either returns the data to the end application, or retries the read at the leader.

4.3 Failures and View Changes

So far, we described IONIA's normal operation. We now discuss failures and view changes. IONIA is similar to VR with respect to both replica recovery [46, §4.3] and view changes [46, §4.2]. The only difference stems from IONIA's durability logs

```
1: procedure LEADER_READ(k)
2:     if pending request for k in durability_log then
3:         trigger sync ordering and execution
4:         wait till updates are executed
       return store.read(k)
```

```
1: procedure FOLLOWER_READ(k)
       return (store.read(k), applied_index)
```

```
1: procedure META_QUERY(k)
2:     if pending request for k in durability_log then
3:         trigger sync ordering and execution
4:         wait till updates are executed
5:         return (flag=data, value=store.read(k))
6:     if history.contains(k) then
7:         return (flag=index, value=history[k])
8:     return (flag=index, value=lti)          ▷ last trimmed index
```

```
1: procedure CLIENT_READ_AT_FOLLOWER(k)
2:     invoke_parallel(f_res = FOLLOWER_READ(k), l_res =
       META_QUERY(k))
3:     if l_res.flag == data then
4:         return l_res.value        ▷ leader returned actual result
5:     a = f_res.value                ▷ follower's applied index
6:     i = l_res.value                ▷ latest index that modified k
7:     if a ≥ i then                  ▷ follower's result is valid
8:         return f_res.value
9:     return LEADER_READ(k)          ▷ invalid; retry at leader
```

Figure 7: **IONIA Reads.** *Summary of* IONIA*'s read protocol.*

that VR doesn't have. IONIA borrows durability logs from Skyros [28] and thus it inherits Skyros' recovery and view-change. We give a brief overview of these procedures.

In IONIA, when a replica recovers, in addition to recovering the consensus log from the leader of the latest view, a replica must also recover its durability log. This is straightforward: the leader sends its durability log (along with the consensus log as in VR) and the replica sets its durability log as the one sent by the leader. This is correct because the leader's durability log contains completed writes in the correct order.

A view change happens when the leader fails. The main challenge is that the new leader must recover the requests in the old leader's durability log and in the correct linearizable order. This is where supermajority quorums help. To see why, consider an *incorrect* protocol where updates are acknowledged after writing to a majority of durability logs. Suppose that update a completes after which b starts and completes (i.e., $a \rightarrow b$). Let D_i be the durability log of replica S_i. Then, a possible state is D_1:[ab], D_2:[ab], D_3:[ab], D_4:[ba], D_5:[ ]. Now, if S_1 (the current leader) and S_2 fail, then it is impossible to determine the correct order from the remaining durability logs. Writing to a supermajority ensures that, after f failures, at least $\lceil f/2 \rceil + 1$ (i.e., a majority within any available majority) will have the requests in the correct order.

During view change, a new leader in IONIA contacts a majority, collecting the requests in their durability logs. It then constructs the set of acknowledged writes, i.e., requests that are present on at least $\lceil f/2 \rceil + 1$ logs. The leader next establishes the order: for every pair of requests a, b, it examines if b appears after a on at least $\lceil f/2 \rceil + 1$ logs. If so, then it concludes b follows a. Such pairwise dependencies are added to a DAG G; IONIA produces the total order by topologically sorting G. These steps are similar to Skyros [28, §4.6]. In addition, the new leader in IONIA constructs the history over its consensus log and sets lti to the index of the first log entry.

Fallback Path. To ensure availability in cases where a supermajority is not available (but a bare majority is), IONIA falls back to a slow mode where writes are acknowledged only after being synchronously ordered on a majority in 2 RTTs.

4.4 Correctness Proof Sketch

Two conditions must hold for linearizability. **P1.** Writes must respect linearizable ordering. **P2.** Reads must never expose stale data. We provide a proof sketch of P1 and P2.

P1. During normal operation, the leader's durability log is guaranteed to have the completed writes in the correct order because a leader response is required for a write to be considered complete. The leader adds requests to the consensus log from its durability log in order; thus, the consensus log captures the linearizable order. IONIA replicas execute ordered writes in parallel. However, for correctness, conflicting writes must be executed serially and in the same order across replicas. The consensus log establishes a total order of writes. Conflicting writes are executed in the order in which they appear in the consensus log. The consensus logs are identical across replicas (this is ensured by base VR). Therefore, conflicting writes are executed in the same order across replicas.

When the current leader fails, a view change occurs, and the leader of the new view must recover the latest consensus log and the durability log. The recovered logs must reflect the correct linearizable order. This recovery procedure in IONIA is the same as Skyros, which ensures that the new leader reconstructs the correct ordering of writes in the both durability and consensus logs [28, §4.7].

P2. First, reads can go to the leader. Because the leader is guaranteed to have seen all completed writes, and because IONIA checks the durability log for pending writes, reads at the leader are guaranteed to see the latest data. We next discuss the correctness of reads performed at followers.

For linearizability, a read r to a key k must see the effects of all writes to k that completed before r started. Let w be the latest write to k. There are two cases to consider.

C1. First, w could have just completed but not executed yet. In this case, w will be present in the leader's durability log. The meta query will thus catch the pending write w in the durability log and return the latest data after execution. The client check ensures that r sees the latest data as it ignores the follower result in this case.

C2. If w was executed, then w may not be in the durability log and will be present in the leader's consensus log and history. Let w_i be the index of w in the consensus log; since

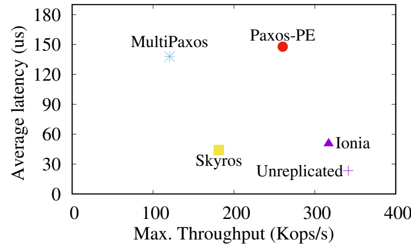


Figure 8: **Write-only Workload.** *The figure plots the maximum throughput and the average latency for a write-only workload;* IONIA, PAXOS-PE*, and unreplicated use 8 threads. The workload loads 350M records.*

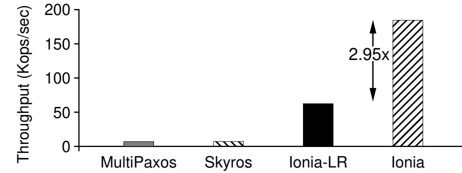


Figure 9: **Read-only Throughput.** *Maximum throughput in MultiPaxos, Skyros,* IONIA-LR*, and* IONIA *under a uniform read-only workload.*

w is the latest write to k, w_i is k's latest modified index. For correctness, the index result returned by the leader for a meta query, res_i, has to be at least w_i, i.e., $res_i \geq w_i$. If the history was not trimmed, then k will be part of the history and the leader will return w_i. Instead, if the history was trimmed then k will not be part of the history. However, we argue why res_i will be greater than or equal to w_i even in this case.

Suppose there were m additional writes independent (i.e., non-conflicting) of w were executed and also the history was trimmed up to $w_i + m$. In this case, leader's last trimmed index, lti, will be $w_i + m$. When the leader doesn't find k in the history, it will return its $lti = w_i + m$, which is greater than w_i. If there were no additional writes and the leader trimmed up to w_i, then the leader's $lti = w_i$, which is safe.

We have established that the leader returns the correct index for a meta query. For final correctness, stale results from a follower must be correctly identified. If w_i was not applied at a follower, the applied-index, a, returned by the follower will be less than w_i. Since the leader's $res_i \geq w_i$, the client check will correctly discard the stale result from the follower.

4.5 Model Checking

We have model checked IONIA's request-processing and view-change protocols. We focus our discussion on IONIA's read protocol because write processing and view changes are similar to Skyros. We generated and explored over 8M different states (e.g., followers having applied up to different points, the leader's history being trimmed up to different points). Linearizability was met in all states.

Our checker finds violations when we intentionally introduce bugs. For example, we modified the model to skip the durability-log check before querying the history upon a meta query. This is unsafe because there could be a completed (but unexecuted) update V_2 in the durability log and follower could have only applied V_1. The leader will return the modified index of V_1, making the client incorrectly trust V_1. Our checker catches this violation. Similarly, the checker identifies a violation when the leader returns indexes lower than the modified index. Finally, we modified the model to have the leader perform the check instead of the client. In this incorrect version, the leader compares the follower's applied-index (a) and the modified index (i) and indicates to the client that data can be trusted if $a >= i$. However, this is unsafe: a client could read the stale version and by the time its meta query reaches the leader, the history could be updated. The leader would then incorrectly indicate to the client that the read is valid. Our checker catches this violation as well.

4.6 Implementation

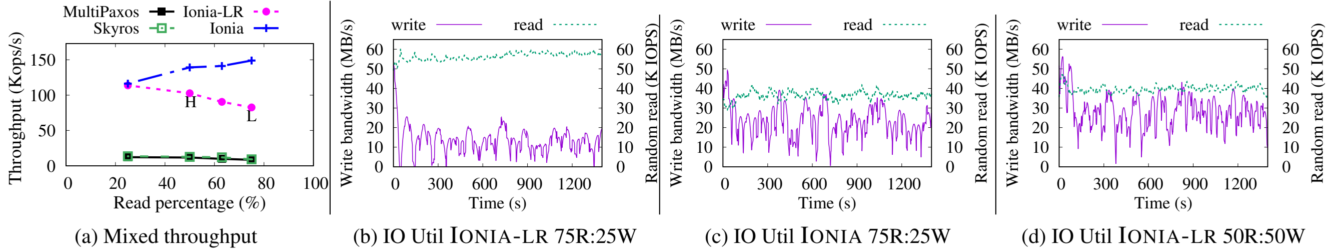
We provide key implementation details. IONIA replicas run SplinterDB [73] as the state machine and communicate via eRPC [36]. We implemented the upcalls in a wrapper, requiring no changes to SplinterDB. IONIA implements a highly concurrent durability log. The IONIA replication layer manages all threads, balancing foreground work (e.g., adding writes to durability logs) and background work (i.e., applying writes to SplinterDB). The leader maintains highly concurrent and compact inverse maps to lookup the durability log and history.

5 Evaluation

To evaluate IONIA, we ask the following questions:

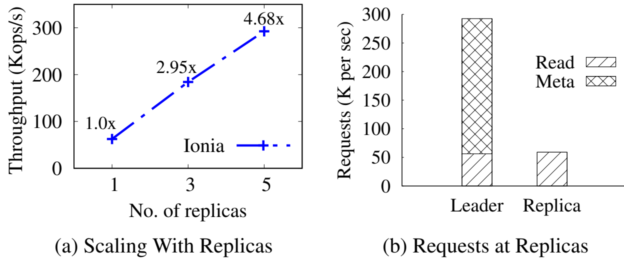
- How does IONIA perform compared to various existing replication approaches for write-only workloads? (§5.1)
- How does IONIA perform for read workloads? (§5.2)
- Does IONIA improve read and also overall performance (by taking off leader reads) under mixed workloads? (§5.3)
- Does IONIA scale throughput with replicas? (§5.4)
- Does IONIA offer low-latency reads? (§5.5)
- How does history size impact IONIA performance? (§5.6)
- How does IONIA perform on the YCSB benchmark? (§5.7)
- Does IONIA improve performance over unreplicated server for read and mixed workloads? (§5.8)
- How does IONIA perform under failures? (§5.9)

Setup. We run our experiments on Cloudlab with three replicas. We compare against: 1. off-the-shelf MultiPaxos (equivalently VR [46]), 2. Skyros [28], a recent low-latency protocol, 3. MultiPaxos with parallel execution (we build this baseline based on CBASE [42]) which we refer to as Paxos-PE, 4. IONIA-LR, an IONIA variant where reads are restricted to the leader, and 5. an unreplicated (fault-intolerant) server. All baselines use batching wherever possible to improve throughput. IONIA replicates SplinterDB. We integrated SplinterDB as the state machine in all baselines. Each replica uses an Intel DC S3520 SATA SSD. Unless specified, we use a 670M KV-pair dataset with 24B keys and 100B values. SplinterDB uses a 4GB cache (as prescribed [73]). For fairness, we modified all baselines to also use eRPC. Unless specified, MultiPaxos-PE, IONIA, IONIA-LR, and unreplicated use 15 threads.



(a) Mixed throughput (b) IO Util IONIA-LR 75R:25W (c) IO Util IONIA 75R:25W (d) IO Util IONIA-LR 50R:50W

Figure 10: **Mixed workloads.** *(a): throughput at various read fractions; (b)-(d): IO utilization of leader in* IONIA-LR *and* IONIA *at two read-write points.*



(a) Scaling With Replicas (b) Requests at Replicas

Figure 11: **Read Scaling.** *(a): read-only throughput with varying replicas. (b): meta queries vs. read requests at* IONIA*'s leader with five replicas.*

5.1 Write-Only Workload

We first analyze write-only performance of MultiPaxos, Skyros, Paxos-PE, and IONIA by comparing them to the an unreplicated server. For each system, we vary the number of clients and measure the maximum throughput and the corresponding average latency. Figure 8 shows the result. We make the following observations. First, as expected, unreplicated server offers high throughput and low latency. Second, MultiPaxos offers significantly lower throughput due to single-threaded execution; further, coordination in the critical path and batching increases MultiPaxos's latencies notably (up to 5.8×). Third, while Paxos-PE is able to achieve higher throughput by via parallel execution, its latency is still significantly higher compared to unreplicated (up to 6.3 ×). Skyros is able to closely match the low latency of unreplicated, but it still offers low throughput similar to MultiPaxos.

Overall, no existing protocol is able to match the high write performance of unreplicated. IONIA, in contrast, closely matches the unreplicated server's high throughput (via parallel execution) and low latency (by deferring ordering and execution to background). This shows that the fault tolerance provided by IONIA comes at no to little performance cost.

5.2 Read-Only Workload

We next analyze the performance of MultiPaxos, Skyros, IONIA-LR, and IONIA for a uniform read-only workload. Figure 9 shows the maximum throughput for each system. Standalone SplinterDB's read performance is limited by random IOPS. However, MultiPaxos use a single thread for executing reads and thus cannot generate the high queue depths needed to saturate SSD IOPS, resulting in lower throughput. Skyros suffers from the same low performance as MultiPaxos due to its single-threaded execution. IONIA-LR executes reads on multiple threads, thus achieving higher throughput, but measuring one level deeper reveals that the IOPS of leader's SSD is saturated. Thus, adding more load after this point doesn't yield higher throughput in IONIA-LR. In contrast, IONIA distributes the reads and extracts bandwidth of followers' SSDs too, providing 2.95× higher throughput than IONIA-LR, achieving essentially linear scaling with 3 replicas. We note here that the leader in IONIA performs meta queries upon every follower read, but this does not create a bottleneck because the meta queries are served from leader's memory.

5.3 Mixed Write-Read Workload

We now analyze mixed read-write workloads. Figure 10(a) shows the maximum throughput of MultiPaxos, Skyros, IONIA-LR, and IONIA for different read percentages with a uniform workload. MultiPaxos offers significantly lower performance than IONIA-LR. Although Skyros commits writes in 1RTT, it achieves only low throughput due to single-threaded execution. As reads increase, IONIA-LR's performance decreases. This is due to IONIA-LR's performance asymmetry (see Figure 2) as all reads go to the leader. IONIA improves performance in two ways by distributing reads. First, reads are served by all replicas, improving read throughput. Second, offloading some reads moves the leader to a more performant regime. IONIA has higher benefits with higher read fraction as more reads can be offloaded. For example, at 75%R, 25%W, IONIA offers 1.8× better throughput than IONIA-LR.

We show how the leader moves into a more performant regime by considering the 75%R, 25%W point. Figure 10(b) and (c) show the leader's IO utilization in IONIA-LR and IONIA, respectively, for this point. As shown in 10(b), IONIA-LR's leader serves more reads than writes; this is a low-performance operating point (denoted as *L* in 10(a)). IONIA splits the 75% reads across 3 replicas, and thus the leader processes equal amount of reads and writes (i.e., 50%-50%). This operating point of IONIA shown in 10(c) roughly resembles the 50%R, 50%W point of IONIA-LR's leader shown in 10(d), a more performant operating point (denoted as *H* in 10(a)).

5.4 Scaling Reads with Replicas

We next show that IONIA can scale reads for cluster sizes widely used in practice [34]. We run a read-only workload with 3 and 5 replicas. Figure 11(a) shows that the through-

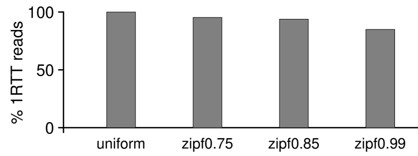


Figure 12: **Fast Reads.** *1-RTT reads for different distributions.*

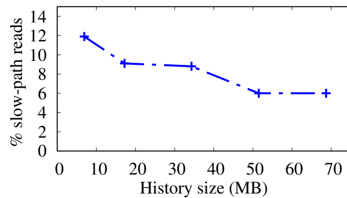


Figure 13: **Impact of History Size on Read Performance.**

put scales linearly. We examine the 5-replica case in more detail. Figure 11(b) shows the overall request throughput at the leader and the average throughput of followers for the 5-replica case. Although the leader receives 4× more load, IONIA is able to scale reads without the leader becoming the bottleneck because the leader serves this 4× greater load (meta queries) cheaply from its memory. Examining the IO utilization reveals that the IOPS on all replicas were saturated, indicating that the collective IOPS remains the bottleneck.

5.5 Low-Latency Reads

We now show that IONIA offers fast reads under different request distributions. A read may take more than 1RTT in two ways. First, a read of key k performed at the leader could trigger synchronous ordering and execution (because of a pending update to k). Second, a client may detect that the follower's result is stale and retry the read at the leader. Both these cases would happen more often with more skewed workloads. Figure 12 shows fraction of 1RTT reads for a read-write (50%-50%) mixed workload with different distributions. As expected, we observe no conflicts for uniform, and thus almost all reads finish in 1RTT. With a zipfian workload (skew factor $\theta = 0.75$ and $\theta = 0.85$), we see a slight decrease as some reads take more RTTs. However, even with a very skewed workload ($\theta = 0.99$), 85% of reads finish in 1RTT.

5.6 Impact of History Size

We next analyze the impact of history size. Intuitively, with a large history, the meta query will more often return the accurate last-modified index. With smaller histories, the leader may often return the last-trimmed index and thus more client checks will fail, resulting in many reads taking the slow path. To study this, we run a read-write (50%-50%) workload with zipfian distribution for various history sizes. As shown in Figure 13, as expected, slow reads decrease with larger histories. However, to achieve good performance in our experimental setup, a 50-MB history is sufficient, a meager 0.06% of the dataset (which contains 670M KV-pairs of 124B each, totalling about 77GB).

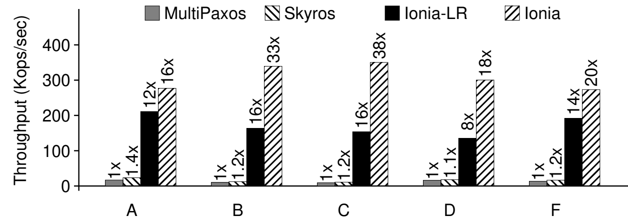


Figure 14: **YCSB Benchmark.** *Throughput under YCSB workloads. Number on top of each bar shows the performance normalized to MultiPaxos.*

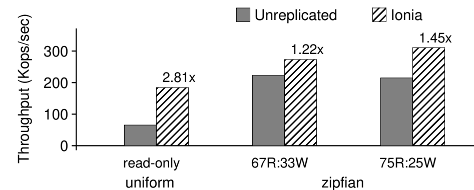


Figure 15: **IONIA vs. Unreplicated: Read and Mixed Workloads.**

5.7 YCSB Benchmark

We now show performance under YCSB [19] workloads: A (50%W, 50%R), B (5%W, 95%R), C (readonly), D (5%W,95%R), and F (50%RMW, 50%R); all workloads are zipfian except D which is latest. Figure 14 plots the throughput for MultiPaxos, Skyros, IONIA-LR, and IONIA. Under all workloads, MultiPaxos offers only low throughput. While Skyros offers marginally higher throughput than MultiPaxos in some workloads, it offers significantly (an order of magnitude) lower throughput compared to IONIA-LR as IONIA-LR employs parallel execution. IONIA preserves the write performance of IONIA-LR and improves over it by distributing reads. The improvement over IONIA-LR in write-heavy workloads (A and F) is roughly 23% to 42%. However, under read-heavy workloads (B, C, and D), IONIA improves performance by ∼2× over IONIA-LR by distributing the large fraction of reads.

5.8 IONIA vs. Unreplicated: Read and Mixed

Finally, we compare IONIA's performance to that of an unreplicated SplinterDB server under read-only and mixed workloads. As shown in Figure 15, for read-only workloads, IONIA offers 2.8× higher throughput by scaling reads, and for mixed workloads IONIA is 1.22× to 1.45× faster.

5.9 Performance under Failures

In the experiments so far, we demonstrated IONIA's performance without failures. We now show that IONIA's performance and availability are not impacted as long as a supermajority is available. Figure 16 shows the throughput over time for a write-only workload with five replicas. Initially, all replicas are up and IONIA achieves high throughput. After a while, one of the replicas fails; however, since four replicas (i.e., a supermajority) are available, IONIA continues to commit requests in 1RTT, thereby maintaining high performance. In contrast, existing approaches (such as CRAQ [70] and Hermes [39]) that offer 1RTT and scalable reads would suffer

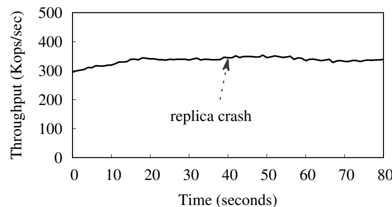


Figure 16: **IONIA Performance Under Failures.**

from write unavailability even when one replica fails.

6 Discussion

Beyond KV stores. In this paper, we focus on designing efficient replication for SSD-based KV stores. However, our ideas apply to other storage systems such as file systems and databases that are built atop write-optimized structures as well. A requirement of our read protocol is that it must be possible to identify which piece of data is read by a request, which holds in many existing systems. Further, while updates in WO-KV stores do not return execution results, other storage systems might support updates that return execution results; however, such updates can be readily supported by IONIA. In particular, when a client issues an update that returns an execution result, IONIA can use the fallback path (described in §4.3) to commit such updates in 2RTT.

Performance with bigger clusters. With larger cluster sizes, a concern might be that meta-queries can become the bottleneck before SSD IOPS. However, this is not a concern in practice today. Most practical systems use only a handful of replicas (3, 5, or 7) [34] and as shown in §5.4, IONIA scales well for such practical cluster sizes.

Performance with slower networks. In most storage systems, SSD IOPS becomes bottlenecked prior to network/NIC. We expect this trend to hold: as SSD IOPS increases, so would the network/NIC packet rate and CPU core count. For example, even when each replica can serve tens of millions of reads/second (via many SSDs), latest NICs (Connect-x6) can serve 215M messages/s [51]. Thus, meta-queries would not be the bottleneck in common scenarios. However, in the (uncommon) deployment scenario where the network/NIC is the bottleneck, as an optimization, many meta-queries can be batched on the client side to amortize CPU/NIC overheads. We leave this optimization as an avenue for future work.

7 Related Work

Other High-Throughput Protocols. Apart from the approaches discussed in §2, a few prior approaches enable multi-core replication through deterministic execution [22, 32]. Other prior systems first execute writes on multiple threads and then replicate the resultant state [71], avoiding non-determinism. Such early execution is a mismatch for WO-KV stores that defer execution for performance. Also, unlike these approaches, IONIA hides ordering and execution latency.

Other Benefits over Low-Latency Protocols. As discussed in §2, IONIA offers 1RTT writes like prior low-latency protocols while offering much higher throughput. IONIA *guarantees* 1RTT writes. Speculative [41, 61] and commutative [44, 53, 59] protocols, in contrast, incur additional RTTs when speculations fail or when writes do not commute. Skyros [28] is a recent protocol that guarantees 1RTT for blind writes by similarly deferring execution. However, Skyros (and the above prior protocols) do not improve throughput or read scaling while reducing latency.

Reading at Non-Leader Replicas. Apart from the protocols discussed in §2, a few prior systems allow reads from a quorum of followers [3, 16]. Shared registers [4] also allows reads at a quorum. While these approaches avoid reads at the leader, they cannot linearly scale read throughput as reads must contact a quorum. Applications over shared logs [5–7] can scale reads by checking the current tail of the log. However, in these approaches, both reads and writes incur high latency, unlike IONIA. Quorum leases [54] maintains per-object leases to scale reads. However, it suffers from high write latency. Also, it is impractical to maintain per-object leases for large disk-based stores. Finally, recent approaches can scale reads through in-network conflict checking [69, 76] or specialized transport protocol [40]. However, these approaches require specialized network hardware (e.g., programmable switches).

Leaderless Approaches. In in-memory replicated systems, message processing overhead at the leader is often the bottleneck [53, 61]. To address this, prior work has built leaderless protocols [49, 53] where any replica can process writes. In a disk-based replicated storage system, however, the leader's storage throughput is the bottleneck. IONIA improves this throughput by batching writes in the background and also taking off read requests from the leader.

LSMs in Distributed Systems. Many prior efforts have optimized LSMs in distributed systems. However, their goals are different from ours, aiming to optimize compactions [29, 72], storage management [75], and load balance [10].

8 Conclusion

We present IONIA, a new replication protocol suited for modern WO-KV stores. IONIA exploits the unique characteristics of WO-KV stores to achieve high performance. We experimentally show that IONIA offers high-throughput, 1RTT writes, and scalable, 1RTT reads. Given that WO-KV stores are widely used, IONIA offers a way to make these stores fault-tolerant with almost no overhead and scale reads.

Acknowledgments

We thank Patrick P. C. Lee (our shepherd) and the anonymous FAST '24 reviewers for their insightful comments and suggestions. We thank the members of DASSL for their discussions. We also thank CloudLab [63] for providing a great environment to run our experiments.

References

[1] Eduardo Alchieri, Fernando Dotti, and Fernando Pedone. Early Scheduling in Parallel State Machine Replication. In *Proceedings of the ACM Symposium on Cloud Computing*, 2018.

[2] Apache. Cassandra. http://cassandra.apache.org/.

[3] Vaibhav Arora, Tanuj Mittal, Divyakant Agrawal, Amr El Abbadi, Xun Xue, Yanan Zhi, and Jianfeng Zhu. Leader or Majority: Why have one when you can have both? Improving Read Scalability in Raft-like consensus protocols. In *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '17)*, Santa Clara, CA, July 2017.

[4] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message-passing Systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[5] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Virtual Consensus in Delos. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '20)*, Banff, Canada, November 2020.

[6] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, CA, April 2012.

[7] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, Pennsylvania, October 2013.

[8] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, July 2019.

[9] Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, Jun Yuan, and Yang Zhan. An Introduction to Be-trees and Write-optimization. *USENIX ;login:*, 40(5):22–28, 2015.

[10] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-Based Databases. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, Lausanne, Switzerland, March 2020.

[11] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos Replicated State Machines As the Basis of a High-performance Data Store. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, MA, April 2011.

[12] Gerth Stølting Brodal and Rolf Fagerberg. Lower Bounds for External Memory Dictionaries. In *SODA*, volume 3, 2003.

[13] Aldenio Burgos, Eduardo Alchieri, Fernando Dotti, and Fernando Pedone. Exploiting Concurrency in Sharded Parallel State Machine Replication. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[14] Tushar D Chandra, Vassos Hadzilacos, and Sam Toueg. An Algorithm for Replicated Objects with Efficient Reads. In *Proceedings of the 35th ACM Symposium on Principles of Distributed Computing (PODC '16)*, Chicago, IL, July 2016.

[15] 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. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, November 2006.

[16] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. Linearizable Quorum Reads in Paxos. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '19)*, Renton, WA, July 2019.

[17] CockroachDB. Pebble. https://github.com/cockroachdb/pebble.

[18] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, Online, July 2020.

[19] 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 '10)*, Indianapolis, IA, June 2010.

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

[21] James Cowling and Barbara Liskov. Granola: Low-overhead Distributed Transaction Coordination. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA, June 2012.

[22] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos Made transparent. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, pages 105–120, Monterey, California, October 2015.

[23] Effi Ofer, Danny Harnik, and Ronen Kat. Object Storage Traces: A Treasure Trove of Information for Optimizing Cloud Workloads. https://www.ibm.com/cloud/blog/object-storage-traces.

[24] Ian Aragon Escobar, Eduardo Alchieri, Fernando Luís Dotti, and Fernando Pedone. Boosting Concurrency in Parallel State Machine Replication. In *Proceedings of the 20th International Middleware Conference*, 2019.

[25] Facebook. Merge Operator. https://github.com/facebook/rocksdb/wiki/Merge-Operator.

[26] Facebook. RocksDB. http://rocksdb.org/.

[27] Pedro Fouto, Nuno Preguiça, and João Leitão. High Throughput Replication with Integrated Membership Management. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, July 2022.

[28] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting Nil-Externality for Fast Replicated Storage. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)*, Virtual, October 2021.

[29] Panagiotis Garefalakis, Panagiotis Papadopoulos, and Kostas Magoutis. ACaZoo: A Distributed Key-Value Store Based on Replicated LSM-Trees. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, pages 211–220, 2014.

[30] Vasilis Gavrielatos, Antonios Katsarakis, and Vijay Nagarajan. Odyssey: The Impact of Modern Hardware on Strongly-Consistent Replication Protocols. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys '21)*, Online, April 2021.

[31] Sanjay Ghemawhat, Jeff Dean, Chris Mumford, David Grogan, and Victor Costan. LevelDB. https://github.com/google/leveldb, 2011.

[32] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: Replication at the Speed of Multi-core. In *Proceedings of the EuroSys Conference (EuroSys '14)*, Amsterdam, The Netherlands, April 2014.

[33] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3), July 1990.

[34] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[35] Intel. Intel Optane 905P. https://www.intel.com/content/www/us/en/products/sku/129833/intel-optane-ssd-905p-series-1-5tb-12-height-pcie-x4-20nm-3d-xpoint/specifications.html.

[36] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI '19)*, Boston, MA, February 2019.

[37] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All About Eve: Execute-verify Replication for Multi-core Servers. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, CA, October 2012.

[38] Karthik Ranganathan. Low Latency Reads in Geo-Distributed SQL with Raft Leader Leases. https://blog.yugabyte.com/low-latency-reads-in-geo-distributed-sql-with-raft-leader-leases/.

[39] Antonios Katsarakis, Vasilis Gavrielatos, MR Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay 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 '20)*, Lausanne, Switzerland, March 2020.

[40] Marios Kogias and Edouard Bugnion. HovercRaft: Achieving Scalability and Fault-Tolerance for Microsecond-Scale Datacenter Services. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys '20)*, Heraklion, Greece, April 2020.

[41] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

[42] Ramakrishna Kotla and Michael Dahlin. High Throughput Byzantine Fault Tolerance. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '04)*, Florence, Italy, June 2004.

[43] Leslie Lamport. Paxos Made Simple. *ACM Sigact News*, 32(4):18–25, 2001.

[44] Leslie Lamport. Generalized Consensus and Paxos. 2005.

[45] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[46] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. 2012.

[47] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Pacific Grove, CA, October 1991.

[48] LogCabin. LogCabin. https://github.com/logcabin/logcabin.

[49] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, CA, December 2008.

[50] Mark Callaghan. Types of Writes. http://smalldatum.blogspot.com/2014/04/types-of-writes.html.

[51] Mellanox. ConnectX-6 Card. https://support.mellanox.com/s/productdetails/a2v50000000p8ReAAI/connectx6-card.

[52] MongoDB. Read Concern Linearizable. https://docs.mongodb.com/manual/reference/read-concern-linearizable/.

[53] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, Pennsylvania, October 2013.

[54] Iulian Moraru, David G Andersen, and Michael Kaminsky. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*, Seattle, WA, November 2014.

[55] Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, 2014.

[56] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, June 2014.

[57] Oracle. Berkeley DB. http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html.

[58] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4), 1996.

[59] Seo Jin Park and John Ousterhout. Exploiting Commutativity For Practical Fast Replication. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI '19)*, Boston, MA, February 2019.

[60] Percona. Percona TokuDB. https://www.percona.com/software/mysql-database/percona-tokudb.

[61] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI '15)*, Oakland, CA, May 2015.

[62] 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.

[63] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), 2014.

[64] Robert Escriva. HyperLevelDB. https://github.com/rescrv/HyperLevelDB.

[65] Sarang Masti. How we built a general purpose key value store for Facebook with ZippyDB. https://engineering.fb.com/2021/08/06/core-data/zippydb/.

[66] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[67] SpeeDB. RocksDB Basics. https://docs.speedb.io/rocksdb-basics#key-tombstones-delete.

[68] Amy Tai, Andrew Kryczka, Shobhit O. Kanaujia, Kyle Jamieson, Michael J. Freedman, and Asaf Cidon. Who's Afraid of Uncorrectable Bit Errors? Online Recovery of Flash Errors with Distributed Redundancy. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, July 2019.

[69] Hatem Takruri, Ibrahim Kettaneh, Ahmed Alquraan, and Samer Al-Kiswany. FLAIR: Accelerating Reads with Consistency-Aware Network Routing. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI '20)*, Santa Clara, CA, February 2020.

[70] Jeff Terrace and Michael J Freedman. Object storage on craq: High-throughput chain replication for read-mostly workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '09)*, San Diego, CA, June 2009.

[71] Robbert Van Renesse and Fred B Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, December 2004.

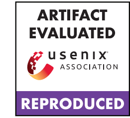
[72] Michalis Vardoulakis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tebis: Index Shipping for Efficient Replication in LSM Key-Value Stores. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys '22)*, Rennes, France, April 2022.

[73] VMware. SplinterDB. https://splinterdb.org/.

[74] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: Separating Data and Metadata for Efficient and Available Storage Replication. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA, June 2012.

[75] Qiang Zhang, Yongkun Li, Patrick PC Lee, Yinlong Xu, and Si Wu. DEPART: Replica Decoupling for Distributed Key-Value Storage. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST '22)*, Santa Clara, CA, February 2022.

[76] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan Ports, Ion Stoica, and Xin Jin. Harmonia: Near-Linear Scalability for Replicated Storage with In-Network Conflict Detection. 13(3):376–389, nov 2019.



Physical vs. Logical Indexing with IDEA: Inverted Deduplication-Aware Index

Asaf Levi*, Philip Shilane†, Sarai Sheinvald§, Gala Yadgar*
*Computer Science Department, Technion †Dell Technologies §Braude College of Engineering

Abstract

In the realm of information retrieval, the need to maintain reliable term-indexing has grown more acute in recent years, with vast amounts of ever-growing online data searched by a large number of search-engine users and used for data mining and natural language processing. At the same time, an increasing portion of primary storage systems employ data deduplication, where duplicate logical data chunks are replaced with references to a unique physical copy.

We show that indexing deduplicated data with deduplication-oblivious mechanisms might result in extreme inefficiencies: the index size would increase in proportion to the logical data size, regardless of its duplication ratio, consuming excessive storage and memory and slowing down lookups. In addition, the logically sequential accesses during index creation would be transformed into random and redundant accesses to the physical chunks. Indeed, to the best of our knowledge, term indexing is not supported by any deduplicating storage system.

In this paper, we propose the design of a deduplication-aware term-index that addresses these challenges. *IDEA* maps terms to the unique chunks that contain them, and maps each chunk to the files in which it is contained. This basic design concept improves the index performance and can support advanced functionalities such as inline indexing, result ranking, and proximity search. Our prototype implementation based on Lucene (the search engine at the core of Elasticsearch) shows that IDEA can reduce the index size and indexing time by up to 73% and 94%, respectively, and reduce term-lookup latency by up to 82% and 59% for single and multi-term queries, respectively.

1 Introduction

One of the most effective ways to address growing storage requirements in datacenters is data deduplication: duplicate *chunks* of data are identified and replaced by references to a single unique copy of each chunk. The mechanisms involved in data deduplication have been optimized in numerous studies and commercial systems. As a result, most backup and archival systems [25,80], as well as many *primary* (non-backup) storage systems and appliances [20, 24, 35, 43], currently support data deduplication.

Data deduplication entails a distinction between the user's *logical* data and the *physical* chunks stored in the system. This additional level of abstraction introduces new challenges in data management. The implicit sharing of content between files complicates, for example, garbage collection [39,40,62], load balancing between volumes [30,37,38,49,51], caching [44,55,56], and charge-back [69]. Fragmentation, which results from newly written files referencing a combination of 'old' chunks and newly written chunks, transforms logically-sequential data accesses to random I/Os in the underlying physical media. This has been addressed in the context of file-read and restore performance [33,45,57,63] and in full-system scans [42].

In this paper, we address *keyword indexing*, an important functionality that is supported by many storage systems [17,26,27] and is severely complicated by deduplication. Specifically, we refer to term-to-file indexing (also known as *inverted indexing*), which supports queries that return the files containing a *keyword* or *term*. Inverted indexes are widely used for simple queries, e.g., by users on personal computers, as well as for complex and batch queries involving multiple terms in a large-scale repository, e.g., by search engines [36], data analytics jobs [61,64,70], and legal discovery [67,77]. The searched data might be deduplicated, e.g., in shared file systems, code repositories, or systems storing similar VM images.

Two aspects of keyword indexing are affected by deduplication. The first is initial index creation time: the system is scanned by processing the logical files, generating random accesses to physical chunks. In addition, chunks are processed redundantly when there are multiple references to a chunk due to deduplication. The second aspect is the index size, which is proportional to the logical data size rather than to the physical size stored in the system: each term must point to all the files containing it, even if the files' content is almost identical. The inflated index size can result in poor lookup performance and also overshadow any capacity savings achieved by deduplication.

Indeed, to the best of our knowledge, systems with high deduplication ratios (i.e., a large number of references to each unique chunk) typically *do not support full keyword indexing*. For example, VMware vSphere [25] and Commvault [19] support file indexing, which only identifies individual files within a backup according to their metadata. Dell-EMC Data Protection Search [21] supports full content indexing, but warns that "processing the full content of a large number of files can be time consuming" and recommends performing targeted indexing on

specific backups or file types.

We address these challenges by *deduplication-aware keyword indexing*. We introduce *IDEA*, which replaces the term-to-file mapping in traditional indexes with a term-to-chunk mapping, whose size is proportional to the unique content physically stored in the system. An additional chunk-to-file mapping records references from chunks to the files they are contained within. This 'reverse mapping' is significantly smaller than the term-to-chunk map and can be stored in a smaller and faster storage device such as SSD or NVRAM. IDEA focuses on textual data. It uses a *white-space aware* content-defined chunking algorithm that creates chunk boundaries that align with white-space characters. This ensures that terms are not split between adjacent chunks.

IDEA creates the index by sequentially processing the physical data instead of the logical data. The term-to-chunk mapping is created by standard term-indexing software, which scans all the physical chunks in the system, disregarding their logical order in the containing files. The chunk-to-file mapping is created by scanning the file metadata, which is typically stored separately from the data chunks. Term lookup in IDEA begins with querying the term-to-chunk mapping. The set of resulting chunks is then used for lookup in the chunk-to-file map, producing a set of matching files.

This basic design of IDEA can support additional functionalities provided by traditional indexes, including low-overhead, incremental indexing of incoming data streams as part of their ingestion, ranking of documents with metrics such as TF-IDF, and returning, for each file in the query result, the offsets in which the keywords were found.

We make the following contributions in this paper:

- We identify and demonstrate the challenges involved in indexing deduplicated data.
- We propose IDEA, the first design of a deduplication-aware term index.
- We describe a prototype implementation of IDEA. For this prototype, we integrated Lucene [2], an open-source single-node inverted index software (similar to that used by the distributed Elasticsearch [4]), into the Destor deduplicating storage system [46].
- We compare the performance of IDEA to a naïve, deduplication unaware, index. On datasets of Linux kernel versions and of English Wikipedia archives, IDEA significantly reduced the indexing and lookup times. For the datasets with a high deduplication ratio, it also reduced the index size.

The rest of this paper is organized as follows. Section 2 gives background and surveys relevant related work, and Section 3 identifies the challenges involved in indexing deduplicated data. Sections 4 and 5 describe the design and implementation of IDEA. Our evaluation setup is described in Section 6, and our evaluation results are analyzed in Section 7. We discuss possible extensions of IDEA in Section 8 and conclude in Section 9.

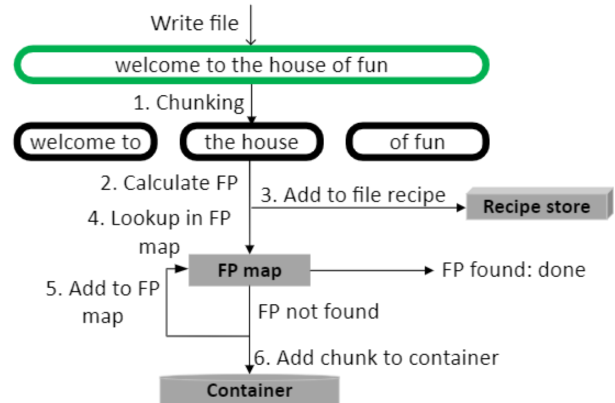


Figure 1: The basic deduplication process.

File	*Content*
F1	Let me into the house
F2	The house of the opera
F3	Welcome to the house of fun

Term	*File (Offset)*
fun	F3(24)
house	F1(16),F2(4),F3(15)
into	F1(7)
let	F1(0)
...	...

Term	*File only*
fun	F3
house	F1,F2,F3
into	F1
let	F1
me	F1
of	F2,F3
opera	F2
the	F1,F2,F3
to	F3
welcome	F3

Figure 2: A toy example of an index of three files.

2 Background and Related Work

Data deduplication. Deduplicating storage systems process incoming data to identify duplicate content and replace it with references to content already stored in the system. Figure 1 gives a schematic view of the main mechanisms of the deduplication process. The data is first split into *chunks* whose average size is typically 4KB-8KB, in a process referred to as *chunking*.

A chunk is represented by a cryptographic hash of its content, referred to as its *fingerprint*. The fingerprint map is queried to determine whether an incoming chunk is already stored in the system. If the chunk is new, it is written and its fingerprint is added to the fingerprint map. Each file is represented by a *file recipe* which contains the file metadata, a list of its chunks' fingerprints, and their sizes. Thus, to read (or *restore*) a file, its recipe is read and its chunks are located by searching in the fingerprint map or a cache of its entries.

The unique physical chunks are written in a log-structured manner, in the order in which they are added to the system. Backup and archival systems usually aggregate chunks, compress them, and pack the compressed data into *containers*, which are the unit of I/O. Containers are several MB in size, and decompression is necessary when restoring chunks. In contrast, deduplication systems for primary storage [34, 41, 43, 72], and especially deduplicating file systems [20, 24], might support direct access to individual chunks.

Keyword indexing. An *inverted index* or a *keyword index*, is a data structure which points from *terms* to their *occurrences*

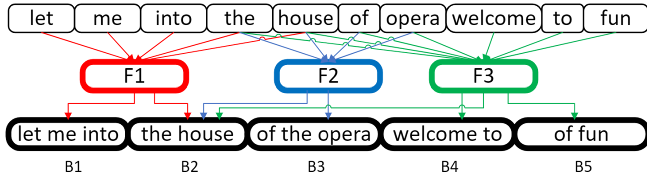


Figure 3: A naive implementation of an index in a deduplicated file system. The resulting inverted index is similar to that in Figure 2.

in a collection of documents. Terms (or *keywords*) can be any searchable strings and are typically natural language words. Any data structure which implements a key-value mapping can be used as an inverted index. From here on, we refer to a keyword index as simply an index, and use *map* to describe individual data structures in the system. Figure 2 shows an example of a small inverted index of a dataset containing three files, where each file is an indexed document. An *index lookup* or *query* returns, for each included term, the list of files containing this term, and optionally the byte offsets in which the term appears. For example, looking up the term “house” in this example will return $\{F_1, F_2, F_3\}$. If offsets are stored, the query will return $\{F_1(16), F_2(4), F_3(15)\}$. Storing the term offsets increases the index size, and is thus supported as an option.

The process of building an index is referred to as *indexing*, and includes the following steps [58]: (1) collecting the documents, i.e., reading the indexed files, (2) identifying the terms within each document, (3) linguistically normalizing the terms (e.g., eliminating plural form and capitalization), and (4) creating the list of documents, and optionally offsets, containing each term. An index can also be built incrementally by a series of *index updates*, where a new set of documents is processed and the existing index is updated to reflect the terms appearing in them. Creating an inverted index is known to be time and resource consuming. Distributing the index and its creation was thus included as a use case in the seminal MapReduce paper [36]. Most index designs support the removal of documents by marking deleted documents and garbage collecting index entries. Some designs avoid the resulting fragmentation in the index structure by batching deletions and rebuilding the index later [2, 58].

Indexing is a key mechanism in information retrieval and presents several challenges that have attracted a wide range of research efforts. One challenge is handling a high rate of incoming new data from sources, such as social media platforms and news services, which needs to be indexed [74, 75], possibly while simultaneously remaining responsive to user queries. Another challenge is supporting not only direct search, but also *similarity search*, in order to provide users with additional related search results [29, 52, 71]. Indexing is used in a wide range of contexts. For example, in natural language processing, an index is used for pre-training and fine-tuning the language model [48, 54]. Indexed pattern matching also plays a key role in bioinformatics [61, 64], data mining [70] and multimedia retrieval [73]. Since the data held in the index itself may be very large [74], an extensive body of work addresses *compressed indexes* with the goal of fitting them in memory [60, 68, 78].

Of available commercial indexing products, the most well-known is Elasticsearch—a distributed search engine supporting full-document indexing and real-time analytics [4, 47]. Elasticsearch is built on top of the single-node Apache Lucene [2]—an open-source full-text search-engine library. Lucene combines a document store with an inverted index that supports searching within any field of the indexed documents, simple lookups, complex queries, analytics jobs, and offsets. Lucene's underlying data structure is based on a hierarchy of skip-lists, which enable sequential access when a query contains multiple terms.

Lucene and its variations serve as the underlying engine of many more commercial indexing products, such as Apache Solr™ [13] and Amazon OpenSearch [1]. IBM Watson [5] is based on distributed Lucene and Indri [6] for indexing large corpora as well as semantic entries and relations between words. Other products support similar document and search interfaces with alternative data structures. For example, Meilisearch [10] is based on LMDB [8] which is implemented with B+ trees, and TypeSense [14] uses the LSM-tree-based RocksDB [12] for its mapping.

Specialized solutions enable search inside compressed structured data such as logs or time-series data. Examples include rapid exhaustive search [22, 23], lazy on-demand indexing of log fields [18], and highly effective in-memory caching of logs [65]. These special-purpose solutions are tightly coupled with the structure of the data and are not directly applicable to the general case of unstructured deduplicated data.

Result ranking. To maximize their relevance, lookup results are typically ranked by index systems, using a *scoring* formula on each result. Among the most popular such formulas, which we use in this paper, is *TF-IDF* [66], used by Lucene (and therefore in Elasticsearch). TF-IDF is commonly defined as follows. Given a document d in which a lookup term t is found, the score TF-IDF(t,d) is defined as $TF(t,d) \cdot IDF(t)$ where $TF(t,d) = \sqrt{\frac{\text{\# occurrences of t in d}}{\text{\# words in d}}}$ and $IDF(t) = 1 + log(\frac{\text{\# docs in the system}}{\text{1+\# docs in which t appears}})$.

Intuitively, TF measures how frequently a term appears in the document, and IDF measures the term significance, based on its occurrence in the entire corpus. Keeping the byte offsets of the terms allows measuring additional attributes such as proximity between multiple terms [32].

3 Challenges

When it was first commercialized, deduplication was primarily applied to backup and archival storage of ‘cold’ data, which is only rarely read and processed [80]. Since then, however, two separate trends have changed the way deduplicated data is accessed. The first is the growing need to process cold data, including old backups. Common scenarios include full-system scans for malware and anomaly detection [53, 76], as well as

keyword searches for legal disclosure [67, 77]. Enterprise applications might perform complex analytics queries, involving multiple-term lookups, on cold storage [31]. In disaster recovery situations, needed VMs may be identified with search terms and then run directly from backup storage until a primary system is restored [28]. These scenarios were addressed in a recent study of deduplication-aware search [42]. However, in the absence of an index, these exhaustive searches might be prohibitively slow.

The second trend is the growing application of deduplication on primary storage of 'hot' and 'warm' data that is accessed regularly [69]. As a result, deduplicating storage appliances are expected to support various functions, including keyword search. For example, users might perform single-term searches for files within their deduplicated personal workstation or their home directory on a deduplicated shared storage partition.

Since indexing software operates at the file-system level, it is unaware of the underlying deduplication at the storage system. A deduplication-unaware (*naïve*) index would thus schematically resemble the structure in Figure 3. The index will map terms to files, independently of how the files' content is stored in the underlying media. However, this oblivious design is inefficient due to the following three challenges.

Challenge 1: index size. The size of the index grows with the number of distinct terms as well as the number of files in the system. In traditional storage systems, this size is roughly proportional to the size of the stored data. As the data size grows, the storage capacity is scaled accordingly, accommodating the growing index. In deduplicated storage, however, the index grows with the logical data, as every new file must be reflected in the terms' document list, even if its content is almost identical to that of files already in the system. An increased index size might also increase the latency of lookups due to logarithmic search complexity and because smaller portions of it will fit into DRAM.

Challenge 2: indexing time. The indexing process scans all the files in the system to create the list of terms in each document and then the list of documents for each term. Performing such a scan on deduplicated data will result in random I/Os for reading the chunks in the order they appear in the files. For example, creating the index in Figure 3 would perform the following series of chunk reads: $[B_1, B_2, B_2, B_3, B_4, B_2, B_5]$. Chunk B_2 is accessed and processed three times, once for every file it is contained in. Furthermore, recall that chunks are read by fetching their entire container or a compression region within a container. Reading chunks in random order might thus cause high read amplification.

Challenge 3: split terms. Although the chunks in our example contain entire words, the chunking process will likely split the incoming data into chunks at arbitrary positions, splitting words between adjacent chunks. Thus, the terms in the beginning or end of a chunk can be correctly identified only when considering the chunks adjacent to it in each file. Therefore, even if the chunk is identified as duplicate, it must be processed in the context of each file that contains it.

As a result of these challenges, to the best of our knowledge, current deduplicating storage systems do not support indexing of their entire content.

4 IDEA

In this section, we describe the design of our deduplication-aware index, IDEA. We begin with an overview of the key concepts, and then describe each component in detail.

4.1 Overview

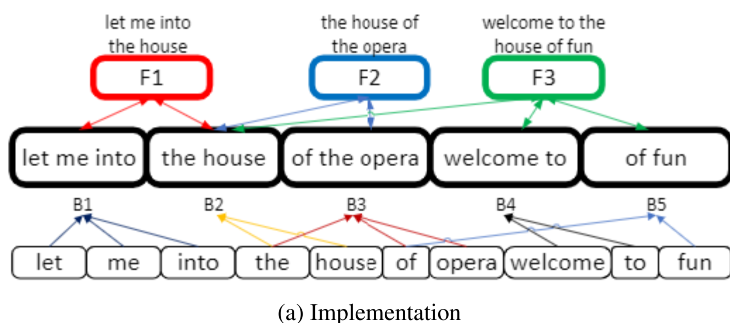
The key idea of deduplication-aware indexing is to map terms to the unique physical chunks they appear in, instead of the logical documents whose number might be disproportionately high. We replace the term-to-file mapping of the traditional index with two complementing maps: a *term-to-chunk map* and a *chunk-to-file map*. The lookup process first finds all the chunks containing the queried terms, and then finds the files containing these chunks. Figure 4(a) depicts the deduplication-aware index that replaces the naïve index in Figure 3. The logical term-to-file mapping from Figure 2 is realized by the combination of the two maps in Figures 4(b) and 4(c), with the file paths resolved by the *file-to-path map* in Figure 4(d). We use file IDs (generated as an internal serial number) in the term-to-file map because they are much smaller than the full file paths. From hereon, we refer to the file IDs as files.

This design allows deduplicating storage systems to provide index functionality to their users. The system can construct The term-to-chunk map with standard indexing software (e.g., Lucene), by passing the chunks as documents for indexing. The chunk-to-file map is based on information from the file recipes, and can be implemented by any standard key-value store. The only modification required in the deduplication system is the chunking process, to ensure that chunk boundaries do not split terms between chunks. We modify the chunking procedure to be *white-space aware* and enforce chunk boundaries only between words.

Properties. The term-to-chunk map is the largest part of the deduplication-aware index. Its size and creation time are proportional to the number of physical chunks. In systems with a high deduplication ratio, this map will be smaller than the term-to-file map in traditional indexing, and will incur lower lookup latency. On the other hand, many optimizations within the traditional index data structures are most effective when files are large (e.g., compressed encoding of file IDs or offsets within files). Processing individual chunks instead of entire files eliminates some of their benefits in a deduplication-aware index. We discuss these cases in detail in Section 7.

4.2 White-space aligned chunking

Deduplication systems employ two types of chunking mechanisms. *Fixed-sized chunking* splits the incoming data into fixed-sized chunks and is typically used in primary storage to align the deduplicated chunks with those of the storage interface. In



(a) Implementation

Term	*Chunk (Offsets)*
fun	B5(3)
house	B2(4)
into	B1(7)
let	B1(0)
me	B1(4)
of	B3(0),B5(0)
opera	B3(7)
the	B2(0),B3(3)
to	B4(8)
welcome	B4(0)

(b) Term-to-chunk

Chunk	*File (Offsets)*
B1	F1(0)
B2	F1(11),F2(0), F3(10)
B3	F2(10)
B4	F3(0)
B5	F3(21)

(c) Chunk-to-file

File	*Path*
F1	home\file1
F2	shared\file3
F3	fun\lyrics\file3

(d) File-to-path

Figure 4: An illustration of a deduplication-aware index (a) and its related maps (b-d) for the files and terms in Figure 2.

	Chunk 1	Chunk 2	Chunk 3
Incoming data	welcome to the house of fun		
Traditional	welcome t	o the hous	e of fun
Whitespace-CDC	welcome to	the house	of fun
Whitespace-fixed	welcome	to the	house of fun

Figure 5: The effect of white-space alignment on chunk content.

other words, the chunks are aligned to the operating-system pages and to the storage-device blocks [20, 24, 34, 43, 56]. *Content-defined chunking (CDC)* splits the data into variable-sized chunks where the hash produced over a rolling window matches a predefined mask. This indicates the start of a new chunk [50, 79, 80]. This method ensures that small differences between similar files will be contained within a small number of chunks and has been shown to achieve better deduplication efficiency than fixed-size chunking [59, 80].

Both techniques are agnostic to word boundaries and will likely end a chunk in the middle of a word. Thus, we modify both to be *white-space aware* and create chunk boundaries that align with white-space and other characters that preserve the locality of words within chunks. These characters are the delimiters used by the indexing software to parse terms during document processing. Thus, white-space awareness is not restricted to a specific encoding or language. In our implementation, the delimiters are defined by the C function `isspace()`.

Content-defined chunking. Systems that use content-defined chunking are designed to handle variable-sized chunks. Thus, extending this mechanism to be white-space aware is relatively straightforward: when it identifies a chunk boundary, instead of immediately triggering a new chunk, we continue scanning the following characters until a white-space character is encountered. This character ends the current chunk and starts the next chunk at the character immediately after it. If a white-space character is not encountered within a sufficiently long distance from the original boundary (512B in our implementation), we leave it unchanged—we assume the split string is irrelevant for indexing anyway.

Fixed-size chunking. Systems that use fixed-size chunking require chunks to fit into fixed-sized memory buffers and/or storage blocks. Thus, if the chunk boundary splits a term in two, we cannot extend this chunk until the end of this term. Instead, when an end of a chunk is identified (by calculating the offset from the beginning of the chunk), we scan this chunk *backwards* until a white-space character is encountered. This character ends the current chunk and starts the next chunk at the character immediately after it. Figure 5 demonstrates the effect of white-space aligned chunking on a small file example.

Although white-space alignment converts fixed-sized chunks to variable-sized chunks, it does not interfere with the deduplication system's operation. The resulting chunks are always smaller than the fixed size, and can thus be stored in a single block (i.e., hard-disk sector or flash page). In addition, recall that file recipes record the size of each chunk, and can thus handle chunks smaller than the fixed size. In case of a deduplicating file system, it can trim the block in memory to the chunk boundary. Additional file system changes might be required to support variable-sized chunks within larger fixed-sized blocks, e.g., recording the chunk size in the inode. These changes are beyond the scope of this project.

Non-textual content. Aligning chunks to white-spaces is only effective in case of textual content, and will have little effect on arbitrary binary content. We thus apply white-space alignment only to chunking of textual content. We identify this content by the file extension of the incoming data, e.g., `.txt`, `.c`, `.h`, and `.htm` files. This distinction during the chunking step allows us to also identify candidate chunks for indexing, and to exclude non-textual content from the indexing process. This is similar to how traditional indexing excludes files based on their extension, e.g., executable files.

The modifications to the deduplication mechanism are minimal. In our implementation, we add a Boolean field to the metadata of each chunk in the file recipe and in its container, indicating whether it is a 'text' chunk or not. During index creation, described in the following subsections, we only process chunks marked as textual.[1] We note, however, that our

[1]We experimented with workloads that contained a mix of textual and non-textual data, and found that the non-textual data does not affect the performance of IDEA. We thus omit those results from our evaluation.

chunking and indexing approaches do not preclude processing of binary content. Non-textual strings are identified by the indexing software (e.g., by the 'tokenizer' in Lucene) and are excluded from the mapping.

Overhead. White-space aligned chunking requires additional processing during the chunking step, and alters the location of chunk boundaries. We verified that enabling white-space awareness increases the chunking time by no more than 0.6% for content-defined chunking. The fixed-size chunking time increased by up to 3×, although it was still only 0.5% of the content-defined chunking time. The resulting number of chunks was within 0.4% and 0.15% of the number of chunks created by content-defined and fixed-size chunking, respectively. The difference in average chunk size was similarly negligible. White-space alignment reduced the deduplication ratio (percentage of data removed by deduplication) of content-defined chunking by no more than 0.4%, and marginally improved that of fixed-size chunking. The experiments were done on the LNX-198 and Wiki-4 datasets, described in Section 6.

4.3 Term-to-chunk mapping

The term-to-chunk map is an inverted index whose documents are physical chunks instead of logical files. The white-space aligned chunking described above ensures that chunks include complete terms, preventing arbitrary prefixes or suffixes from being incorrectly indexed. The number of documents in the index is the number of physical chunks, which might be higher than the number of logical files. The effect of this design choice on the size of the index is evaluated in Section 7.

During indexing, the chunks are read sequentially by fetching entire containers or compression regions, and each chunk is processed only once, regardless of the number of files containing it. Since each chunk is processed independently, processing the chunks is easily parallelizable. We leave related optimizations for future work. A term-lookup in the term-to-chunk map returns the fingerprints of the chunks this term appears in (and optionally its offsets within them). The fingerprints are used for lookup in the chunk-to-file map, described later in this section.

4.4 Chunk-to-file mapping

The mapping from chunks to files is independent of the term-to-chunk mapping, both in structure and in its construction. The mapping is constructed from two complementing maps: in the chunk-to-file map, each chunk fingerprint points to the IDs of all the files that contain this chunk (and optionally its offsets within each file). The file-to-path map connects each file ID to the file's full pathname. This is equivalent to the mapping between document IDs to user-defined document names in traditional inverted indexes.

We implement the chunk-to-file and file-to-path maps as separate key-value stores, whose keys are the chunk fingerprints and file IDs, respectively. Both maps are created from the metadata in the file recipe. For each file, a <fileID,path> pair is added to the file-to-path map, and a <fingerprint,fileID> pair is added to the chunk-to-file map for each fingerprint in the recipe. If the index support offset lookup, then the <fingerprint,fileID> pair also carries the list of offsets in which the chunk appears. This information can be derived from the file recipe, which contains the size of each chunk.

4.5 Keyword/term lookup

IDEA performs keyword lookup in three phases: (1) a lookup in the term-to-chunk map yields the fingerprints of all the relevant chunks and optionally the term offsets within them, (2) a series of lookups in the chunk-to-file map retrieves the IDs of all the files containing these chunks, and optionally the chunk offsets within them, and (3) a lookup of each file ID in the file-to-path map returns the final list of file names. When requested, the offsets of the terms within the files are derived from the combination of the term and chunk offsets.

Phases (2) and (3) are oblivious to the number of keywords in the original search query. The term-to-chunk map, implemented as an inverted index, returns a set of unique chunks, even in complex search queries that lookup multiple keywords. However, when a term appears in multiple chunks belonging to the same file, some of the files returned from the chunk-to-file map will be redundant. When offsets are not supported or not requested in the query, we collect the results from phase (2) in a set data structure that eliminates duplicate entries, ensuring that each file is searched in the file-to-path map only once.

4.6 Ranking results

As a proof of concept, we extended IDEA to support document ranking with the TF-IDF metric. Recall (from Section 2) that the score of a <document,term> pair is calculated using four values. The number of *files in the system* is a global system value. The number of *words in the file* is calculated during index creation: IDEA sums the number of words in each of its chunks, which are counted by indexing software when the chunks are processed.

The remaining values are calculated during the lookup of the term, as follows. The number of *files containing the term* is the number of files in the query result. Calculating the number of *appearances of the term in the file* is equivalent to counting the offsets of this term within the file. If offsets are not supported, IDEA supports ranking by recording number of appearances instead of offsets. The term-to-chunk map records the number of appearances of the term in each chunk, and the chunk-to-file map records the number of appearances of each chunk in the file. These values are combined for the files in the query result, to return the number of appearances of the term in each file.

Supporting ranking for a query with multiple terms requires calculating the number of appearances of each term in each file, separately. This can be done by maintaining a temporary data structure that collects, for each term, the chunks it appears in. This information can be combined with the number of appearances of each chunk in each file in the query result. IDEA can directly support ranking with any metric that is

based on term occurrences or positions. Other metrics may also be supported, but are outside the scope of this paper.

5 Implementation

We implemented a prototype of IDEA to show how deduplicating storage systems can provide indexing functionality with our approach. Storage systems have direct access to file recipes and physical chunks, and can use an existing engine for indexing. In our implementation, we integrated Apache Lucene [2] into the Destor open-source deduplicating backup system [46]. Destor was built for academic purposes and includes all fundamental deduplication mechanisms and data structures. It supports *backup* and *restore* operations, creating deduplicated backups of entire directories. We used the open-source C++ implementation of Apache Lucene, LucenePlusPlus [9], and the C++ version of Destor [3] that was used for deduplication-aware exhaustive scans [42].

Lucene assigns an internal *document ID* to every document that it processes. Its index consists of two major data structures. The *term-to-doc map* returns, for each term, a list of document IDs it is contained in. The *document store* contains, for each document ID, attributes such as its size and file path, and possibly its content. In IDEA, we realize the term-to-chunk map using Lucene's term-to-doc. The document store realizes the chunk-to-file map: the document representing each chunk contains the list of files this chunk is contained in. We use Berkeley-DB [11] for the file-to-path map. It is implemented with the recno data structure, which is a flat text format optimized for sequential integer keys. Figure 6 illustrates the data structures used by Lucene and by IDEA. By default, IDEA uses an SSD for the data structures which are external to Lucene, as Lucene uses the main memory to cache the parts of the index required for fast access. We evaluate the effect of the SSD below.

The full indexing process in IDEA proceeds as follows. We first scan all the file recipes from Destor and create the list of files containing each chunk using a key-value store, which may spill to disk. Each list is added as a document (which is immutable in Lucene[2]) to the document store, where the document ID is the chunk ID. Then, we read the containers from Destor's on-disk container store to memory. The chunks in the containers are passed to Lucene for indexing in the term-to-document map, with their respective IDs.

IDEA must ensure that its separate maps remain consistent. In other words, all the chunks returned from the term-to-chunk map must be present in the chunk-to-file map, and all the file IDs must map to full file pathnames. Thus, IDEA ensures that the <fileID,path> pair is persisted in the file-to-path map before it uses this ID in the chunk-to-file map. IDEA currently does not support lookups to be issued in parallel with index creation. IDEA records, in the file recipe, whether the file has been indexed or not, and a similar record marks containers

[2]The documents are stored sequentially in the index segments, and are immutable to facilitate efficient direct access to them via the skip-lists.

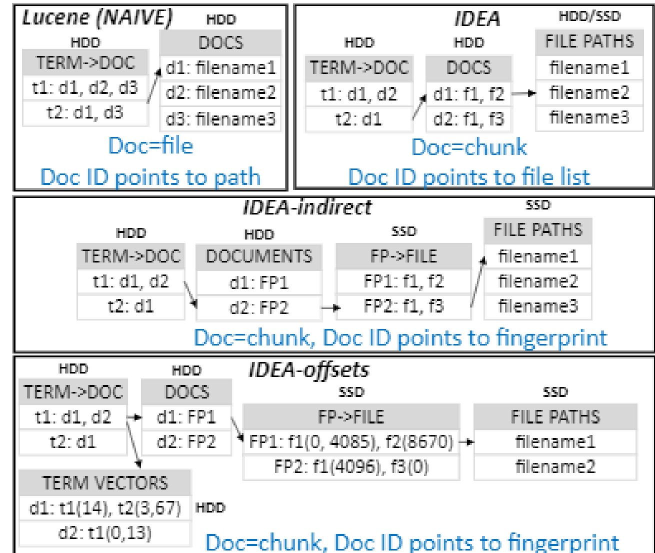


Figure 6: Data structures for Naïve and IDEA, including variants of IDEA that support offsets and ranking.

whose chunks have been indexed. In case of a system crash during index creation, IDEA can be restored to a consistent state by re-processing the unindexed containers and file recipes. Duplicate entries in the term-to-chunk map will be handled by Lucene. In the chunk-to-file map, we can look up each chunk before adding its <chunk,fileID> pair, to find out whether it was already inserted to the map.

We perform the three phases of keyword lookup described in Section 4.5 sequentially. We use the set data structure of C++ (`std::set`, implemented as a red-black tree) to store the unique sets of files returned from chunk-to-file map. Lucene uses a similar structure to return unique file-IDs in the term-to-chunk lookup of multiple keywords. For multiple-keyword lookups, we use Lucene's OR query with all the keywords.

IDEA-indirect. To support additional index functions, we implemented an alternative, more modular, version of IDEA, by adding another level of indirection to its maps. The main advantage of IDEA-indirect is its ability to support *inline indexing*, as part of the system's processing of incoming data: the separation between the document store and the mapping of chunks to files precludes the need to create an immutable list of files containing each chunk.

In IDEA-indirect, the chunk-to-file map is split into two maps: the document store holds, for each document ID (representing a chunk), this chunk's fingerprint. An additional map returns, for each fingerprint, the files containing its chunk. Figure 6 illustrates these data structures. The FP-to-file and file-to-path maps are stored on SSD for faster lookup. We use Berkeley-DB [11] for the additional FP-to-file map. We implement it as a hash-table based multi-map, which supports efficient additions of <chunk,fileID> pairs whenever a new reference to a chunk is identified. In inline indexing, new chunks are passed to Lucene for indexing as soon as they are identified

Dataset	Logical (GB)	Physical (GB)	Recipe (GB)	Files	Chunks (M)
LNX-198	51	11	0.7	4.3M	1.6
LNX-409	181	13	2.4	15.3M	1.7
LNX-662	334	14	4.7	28.2M	1.8
Wiki-4	242	114	0.884	2.3K	10.3
Wiki-8	487	180	1.8	4.7K	14.9
Wiki-12	736	255	2.8	7K	20.1
Wiki-12-1MB	736	259	2.9	686K	20.1
Wiki-24-1MB	1370	478	5.4	1.3M	36.9

Table 1: The datasets used in our experiments

Dictionary	LNX-198		Wiki-12	
	Files	Chunks	Files	Chunks
file-low	1.4	1.3	1.3	1.11
file-med	9.5	3.3	9.35	3.57
file-high	93.7	14.4	95.4	40.1
chunk-low	11.6	1.22	8.25	1.43
chunk-med	28	9.6	16.1	9.36
chunk-high	148	94.8	208.9	94.6

Table 2: Average number of files and chunks per keyword in dictionaries used in our experiments.

as unique (in parallel to step 6 in Figure 1). The <chunk, path> pairs are added as <chunk, fileID> and <fileID, path> to their respective maps after fingerprint calculation, during the creation of the file recipe. Lucene organizes its index in segments, and automatically creates a new segment when new documents are added to an existing index. Segment merging and splitting is controlled by a set of Lucene's internal triggers.

IDEA-offsets. In LucenePlusPlus [9], which is equivalent to Lucene version 3.0.3, term offsets within files are maintained in dedicated data structures called *term vectors*: for each document, the term vector lists the terms in this document, and the offsets each term appears in [3]. We rely on these existing structures to support offset lookups in IDEA-offsets, which extends IDEA-indirect as follows. IDEA-offsets uses the term vectors to record the offsets of terms within the chunks they appear in. It also extends the chunk-to-file map to record, for each file-ID, the list of offsets of the chunk within the file. Figure 6 illustrates these data structures.

IDEA-rank. LucenePlusPlus supports ranking by recording term frequencies within the term-to-doc map. This version of Lucene couples support for ranking with support for proximity search: the term frequency is recorded alongside the list of its positions in the files.[4] This increases the size of the term-to-doc map beyond what is necessary for ranking alone.

We implement IDEA-rank by extending IDEA-indirect to use the frequency records of Lucene. It stores the frequency of the term in each chunk in the term-to-chunk map, and the frequency of the chunk in each file in the FP-to-file map. The number of terms in each file is stored in the file-to-path map, and the global counter of files is maintained as an independent counter. IDEA-rank currently supports a single-term lookup, with multi-term queries deferred to future work.

6 Experimental Setup

Baseline. In addition to IDEA, IDEA-indirect, IDEA-offsets and IDEA-rank, we evaluated a deduplication-oblivious index, **Naïve**, which uses Lucene to index the logical files as documents. To implement Naïve, we extended Destor's restore process: we use it to read all the files in the system in their logical order. Instead of writing the restored files, they are passed to Lucene as documents for indexing. Lookup in Naïve is performed by a simple OR-query lookup. After retrieving the document IDs from the inverted index, Lucene converts them to file names and returns the names as the query result (see Figure 6). We also implemented versions supporting additional functionality, **Naïve-offsets** and **Naïve-rank**.

Datasets. We used two types of datasets for evaluating indexing and lookup times, similar to those used in [42]. The **Linux** datasets contain versions of the Linux kernel source code [7], from version 2.0 to version 5.9. The datasets contain 198, 409, and 662 versions, including all the minor versions, every 10th patch, and every 5th patch, respectively. The **Wikipedia** datasets contain archived versions of the English Wikipedia [15, 16], from January 2017 to March 2018. The datasets contain 4, 8, 12, and 24 consecutive XML dumps, which we split into files of 100MB (at page boundaries). We created two additional datasets from the 12 and 24 versions, with files of 1MB, to evaluate the effect of the number of files on the index performance. The datasets were created with variable-sized chunks (using Rabin fingerprints) with an average size of 8KB. The full details appear in Table 1.

Keyword dictionaries. We created six sets (*dictionaries*) of keywords that vary in the number of chunks and files they appear in. We sorted the terms in Wiki-12 in order of the number of chunks they appear in, and retrieved all the terms that appear in the ranges of 1-2, 9-10, and 90-100 chunks. We then chose 128 random terms from each range, creating the Wiki-chunk-low, Wiki-chunk-med, and Wiki-chunk-high dictionaries. We repeated this process, counting appearances of terms in entire files instead of chunks, to create Wiki-file-low, Wiki-file-med, and Wiki-file-high. We created a similar set of dictionaries from LNX-198 using the same process. The resulting average number of chunks and files containing each term in each dictionary are summarized in Table 2.

Hardware. For our experiments, we used a server running Ubuntu 16.04.7, equipped with 128GB DDR4 RAM and an Intel Xeon Silver 4210 CPU running at 2.40GHz. The backup store for Destor was a Dell 8DN1Y 1TB 2.5" SATA HDD. The maps of all the index alternatives (Naïve as well as the term-chunk map of IDEA and all the maps of IDEA-Direct) were stored on a separate identical HDD. The chunk-to-file and file-to-path maps of IDEA were stored on a Dell T1WH8

[3]Later versions of Lucene also support the embedding of offsets within the term-to-doc map, eliminating the additional data structure.

[4]A position is the term's offset counted in terms, rather than bytes.

240GB 2.5" SSD. We cleared the page cache and restarted Lucene before each indexing and lookup experiment.

7 Evaluation

The goal of our experimental evaluation was to understand how deduplication-aware indexing (IDEA) compares to the traditional deduplication-oblivious indexing (Naïve) in its storage requirements (index size), memory usage, indexing time, and lookup performance. We designed our evaluation to demonstrate how these aspects are affected by the characteristics of the indexed data (deduplication ratio, number and size of files) and of the searched keywords.

Indexing time. Figure 7 shows the offline indexing times of Naïve and IDEA. It shows that deduplication-aware indexing can reduce indexing time compared to Naïve, and that the reduction is proportional to the *deduplication ratio*—the portion of data removed by deduplication. The recipe-processing time is negligible compared to the chunk-processing time in all except very extreme cases (LNX-662 with 28M files).

Recall that the Linux datasets have a very high deduplication ratio (between 78% in LNX-198 and 95% in LNX-662), with many small files. In these datasets, the indexing time of IDEA is shorter than that of Naïve by 76% to 94%. This reduction results from processing each chunk only once, and fetching the chunks sequentially from the underlying HDD. The Wikipedia datasets have lower deduplication ratios (between 53% in Wiki-4 and 65% in Wiki-12 and Wiki-24) and a considerably smaller number of large files. In these datasets, the reduction in indexing time is substantial but smaller: the indexing time of IDEA is shorter than that of Naïve by 49% to 76%.

The results for the Wiki-12 versions further illustrate the effect of the number and the size of the files on the indexing times. When Lucene creates the term-to-file mapping, multiple occurrences of a term in a document are heuristically replaced (in memory) by a single pointer/counter. Thus, as the size of the files decreases (from 100MB in Wiki-12 to 1MB in Wiki-12-1MB), there are fewer replacements and their processing time in Naïve increases. In contrast, the chunk-processing time of IDEA depends on the appearances of terms in chunks, not files, and thus remains similar for all these versions.

Index size. Figure 8 compares the size of the different indexes. In the Linux datasets, IDEA is always smaller than Naïve, and the difference between them increases with the deduplication ratio—for LNX-662, the index size of IDEA is 73% smaller than that of Naïve. The reason is the large number of small files, combined with a very high deduplication ratio: there are more files than chunks in all these datasets (see Table 1). In Naïve, each term points to a large set of files it occurs in, while in IDEA, the files are recorded for each chunk rather than term. The file-to-path map occupies a significant portion of IDEA's index for these datasets. However, it is still considerably smaller (27%-44%) when compared to Naïve.

The Wikipedia datasets have a much lower deduplication ratio than Linux. In this case, the benefit from mapping term to chunks instead of files depends on the size of the files. The index of IDEA is larger than that of Naïve in the datasets with 100MB files (Wiki-4, Wiki-8, Wiki-12), and is smaller in the datasets with 1MB files. The advantage of IDEA compared to Naïve can be seen when comparing the different versions of the Wiki-12 dataset: the size of Naïve grows considerably with the number of files, while the size of IDEA is almost unchanged. The reason is that when the data is split into more files, Naïve must record more files for all the terms included in them. In IDEA, however, this additional information is recorded per chunk, not per term.

The memory requirements of each index are related but not directly proportional to the index size. During startup, Lucene loads an internal data structure called the *term-info-index (Tii)*, which contains statistics regarding each term, including a compressed counter of its frequency in the entire dataset. The size of the Tii is roughly half of the size of the data loaded into memory during Lucene's startup. The Tii in IDEA is smaller than the Tii in Naïve (except in Wiki-4, which has an unusually low deduplication ratio) by 52% to 76%. The peak memory usage of IDEA is proportionately lower than that of Naïve for the Wikipedia datasets. In the Linux datasets, more memory is used for chunk-to-file lookups, and thus its consumption is comparable to that of Naïve.

Lookup times. Figure 9 shows the lookup time for a single keyword from two dictionaries in three representative datasets. Each bar represents an average of four experiments (the standard deviation was at most 0.2%), each with a different keyword, with the latency divided into startup time and lookup times in each map. The results show that the additional lookups due to the indirection in IDEA have a minor effect on the latency. IDEA is faster than Naïve by up to 82%, 47%, and 45% in LNX-198, Wiki-12, and Wiki-12-1MB, respectively.

The advantage of IDEA is the smaller size of its term-to-doc map, which incurs shorter lookup latency. The latency of each step in the lookup process depends on the size of the respective data structure. The startup time, which is dominant when searching for a single keyword, is proportional to the size of the Tii, which is much smaller in IDEA. The lookup times in the different inverted term-indexes is also proportional to their size, due to the logarithmic search complexity in Lucene's skip-lists. The remaining latency is incurred when converting document IDs to names. In the Wikipedia datasets, this specific map of IDEA is larger than that of Naïve by orders of magnitude, but its overall lookup time is still smaller than that of Naïve.

Figure 10 shows the lookup times with increasing numbers of terms from the file-med dictionary. Each bar shows the average time of three independent executions, and the standard deviation was at most 0.09%. As the number of terms increases, the weight of the startup time S_{index} in the overall lookup latency decreases, and the time to convert the document IDs to their names increases. IDEA outperforms Naïve by up to 59%

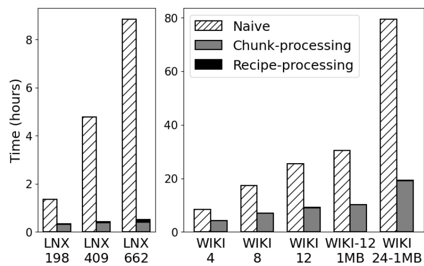


Figure 7: Offline indexing times.

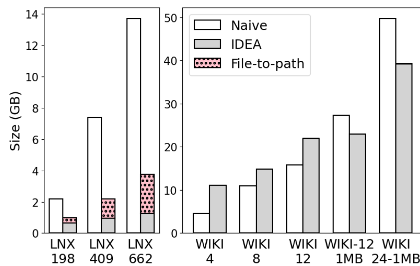


Figure 8: Index sizes.

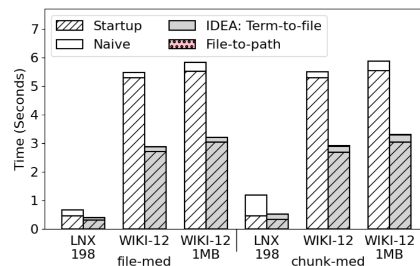


Figure 9: lookup times of a single keyword.

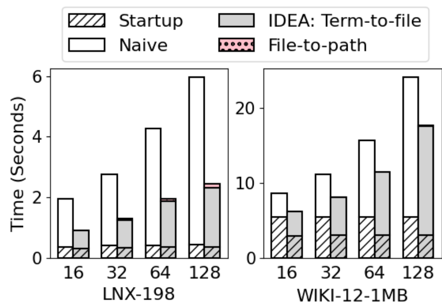


Figure 10: Lookup times with different numbers of keywords, with the file-med dictionary.

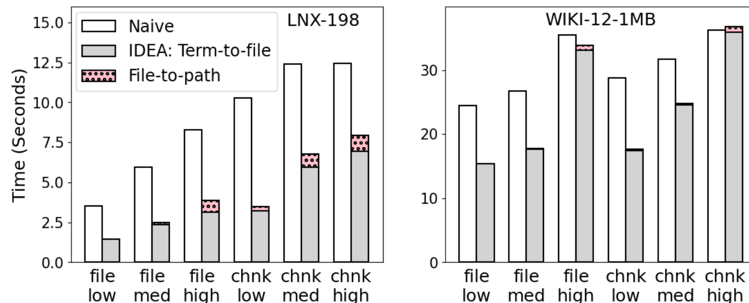


Figure 11: lookup times with 128 keywords of different dictionaries, in the LNX-198 (left) and WIKI-12-1MB (right) datasets.

and 27% on the Linux and Wikipedia datasets, respectively.

Figure 11 compares the effect of different dictionaries on the lookup times. Each bar shows the average time of three independent executions, and the standard deviation was at most 0.09%. The lookup times of Naïve as well as IDEA increased with the number of chunks and files in the query result. The Linux dataset contained more files than chunks, and thus IDEA was faster than Naïve by up to 59% (in the file-low dictionary). The Wikipedia dataset, on the other hand, contained many more chunks than files. As a result, there are considerably more chunks that contain each term, especially in the chunk-high dictionary, for which the lookup time of both indexes was comparable. For all other dictionaries, the lookup time of IDEA was shorter than that of Naïve.

We repeated all the lookup experiments of IDEA without an auxiliary SSD. This means that its file-to-path map was stored on HDD. As expected, this increased the total lookup time, and this increase was proportional to the number of different files in the query result. For example, with a single keyword from the chunk-med dictionary, the increase compared to IDEA with an SSD was 1.1% and 16% in the Wiki-12 and LNX-198 datasets, respectively. The biggest increase was 168% when looking up 128 keywords from the LNX-662 dataset. Nevertheless, the lookup of IDEA was faster than that of Naïve, even without the SSD, in all experiments.

IDEA overheads. We created two datasets to evaluate the worst-case overheads of IDEA in a system without deduplication. LNX-1 and Wiki-1 contain a single version of Linux and Wikipedia, respectively. With almost no deduplication, IDEA has no advantage when compared to deduplication-oblivious indexing, while incurring the overhead of processing a large number of small documents, and looking up terms and chunks in an additional mapping layer. Table 3 lists the characteristics of each dataset, as well as the indexing and lookup times of the different indexes.

Indeed, IDEA is larger than Naïve, due to the larger number of documents in the index: IDEA must record, for each term, all the chunks it appears in, even though many of them point to the same file. The indexing time is similar for both indexes in the Linux dataset that consists of many small files. In the Wikipedia dataset, IDEA cannot optimize index construction by eliminating recurring terms in a document because its documents are small chunks rather than Wikipedia's large files. The lookup times of IDEA are longer for both datasets, due to the reasons discussed in detail above. However, this increase is negligible for Wikipedia and is a modest 10% for Linux.

This experiment emphasizes the tradeoffs of deduplication-aware indexing. Namely, that the additional layer of indirection incurs non-negligible overheads that are masked in systems where the deduplication ratio is sufficiently high. In our future work, we will identify the minimal deduplication ratio for which deduplication-aware indexing is more efficient that the traditional approach, and how this minimum depends on the average file size.

The effect of indirection. IDEA-indirect is the basis for the additional functionality of deduplication-aware indexing (offsets and ranking). To evaluate the effect of the additional layer of indirection (chunk-to-file mapping), we repeated the indexing and lookup experiments with IDEA-indirect. The offline indexing times were within 1% and 6% of those of IDEA for the Wikipedia and Linux datasets, respectively. The size of IDEA-indirect is always larger than that of IDEA: by

Dataset	Dataset size			Indexing time		Index size		Lookup time	
	Logical	Physical	# files / # chunks	Naïve	IDEA	Naïve	IDEA	Naïve	IDEA
WIKI-1	60.4GB	60.3GB	574 / 6M	2.02H	2.25H (11.3%)	2.5GB	6.1GB (152%)	11.67	11.78 (0.1%)
LNX-1	0.91GB	0.86GB	74K / 165K	86 secs	85 secs (N/A)	49MB	60.8MB (24%)	0.47	0.52 (10%)

Table 3: Worst-case overhead of IDEA (in parentheses, with respect to Naïve) for systems without deduplication. The lookup times (in seconds) refer to the average of the file-med and chunk-med dictionaries.

up to 22% and 49% for the Wikipedia and Linux datasets, respectively. Despite this increase, it is still smaller than Naïve in the Linux datasets.

The additional level of indirection also increases the lookup time compared to IDEA. This increase grows with the number of chunks in which the queried keywords appear: in the LNX-198 dataset, the lookup time of 128 keywords with IDEA-indirect was 32% and 23% higher than that of IDEA, in the file-low and chunk-high dictionaries, respectively. In the Wiki-12-1MB dataset and the file-high and chunk-high dictionaries, this increase caused IDEA-indirect to be slower than Naïve. In all other experiments, however, IDEA-indirect was faster than Naïve, despite the additional layer of indirection.

Inline indexing. We compared the inline and offline indexing times of Naïve and IDEA-indirect on two of the datasets, LNX-198 and Wiki-12. Recall that inline indexing is integrated into the deduplication process, referred to as 'backup' in Destor. In this experiment, the original data was read from one HDD and backed-up by Destor on a second HDD. We include the backup time in the results for offline indexing, for a meaningful comparison. Inline indexing is more efficient in terms of memory usage – the chunks are processed while they are still in memory, and do not need to be fetched from the disk. At the same time, the backup process is slowed down by the additional processing.

The results in Figure 12 show that the slowdown of the backup process is detrimental with Naïve indexing: 6.1x and 11.5x in the Linux and Wikipedia datasets, respectively. Indeed, to the best of our knowledge, no deduplicating system currently supports inline indexing. In contrast, IDEA-indirect slows down the backup process by only 1.7x and 4.5x in the Linux and Wikipedia datasets, respectively, thanks to its ability to process only new unique chunks. Although this overhead is not negligible, it presents, for the first time, a realistic opportunity to index deduplicated data inline with writes.

The effect of term offsets. We repeated the indexing and lookup experiments of IDEA with IDEA-offsets. For brevity, we present here only the results of representative datasets and workloads. Figure 13 shows the index size for Naïve-offsets and IDEA-offsets. These sizes are larger than the sizes without offsets, due to the additional information stored in the term vectors. The increase is higher for Naïve than for IDEA: offsets increase the index size by up to 20.9x and 7.1x for Naïve and IDEA, respectively. As a result, the size of IDEA-offset is always smaller than Naïve-offset, even for datasets in which the situation was reversed without offsets (see Figure 8).

The reason for this difference is that the number of offsets stored by Naïve-offset depends on the logical occurrences of each term. This eliminates the "advantage" that Naïve had over IDEA in datasets with large files. In IDEA-offsets, the offsets are recorded only within chunks: their number as well as their values are smaller, occupying less space in the term-vectors. The additional size of the chunk offsets in the FP-to-file is much smaller than that of the term offsets.

Dataset	LNX-198	Wiki-12-1MB
Naïve-rank	10.2GB (332%)	173GB (490%)
IDEA-rank	3.8GB (178%)	80GB (172%)

Table 4: Index sizes with ranking. The number in parentheses is the increase compared to the version without ranking.

IDEA's smaller size also results in faster indexing, shown in Figure 14. Offsets increase the indexing time by up to 51% and 47% for Naïve and IDEA, respectively. The increase is higher for Naïve-offsets due to the larger index that must be persisted on the HDD.

Figure 15 shows the lookup times for 128 words from the different dictionaries for two representative datasets. Lookups with offsets are naturally slower than without them, due to the need to fetch the term vectors for each file in the query result. Comparing the results to those in Figure 11 shows the increase in the lookup time due to the added offsets: the time increased by as much as 33x and 7.5x for Naïve and IDEA, respectively. The increase is higher in Naïve due to different reasons: in LNX-198, it fetches more term vectors from HDD: one for each file the term appears in, rather than one for each chunk. In Wiki-12-1MB, the term vectors are much longer, because each 1MB file contains many more terms than each chunk.

The effect of result ranking. We evaluated the effect of result ranking on two representative datasets, LNX-198 and Wiki-12-1MB. Recall that recording term frequencies in LucenePlusPlus is coupled with the recording of term positions. As a result, the size of both Naïve-rank and IDEA-rank is larger than their versions that do not support ranking. Table 4 shows that this increase is higher for Naïve, similarly to its increased size when offsets are recorded.

The increase in indexing time (shown in Table 5) is milder, and is higher for IDEA than for Naïve. The reason is the additional in-memory processing required for generating the term counters in the file-to-path map of IDEA-rank. Nevertheless, the indexing time of IDEA-rank is still shorter than that of Naïve-rank. Further optimization of the data structures of IDEA-rank is possible and is left for future work.

Figure 16 shows the lookup times with one keyword (averaged over four runs with different keywords) from the file-med

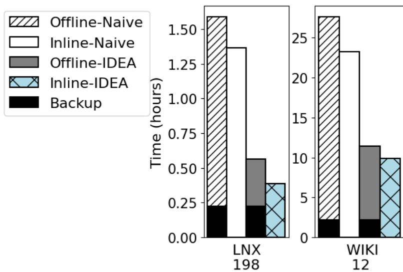


Figure 12: Inline indexing times.

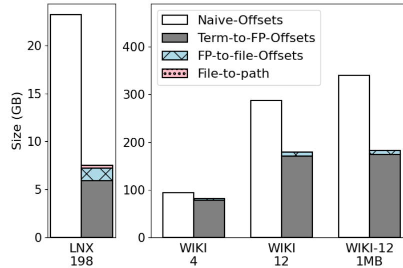


Figure 13: Index sizes with offsets.

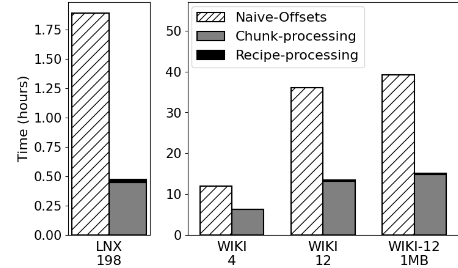


Figure 14: Offline indexing times with offsets.

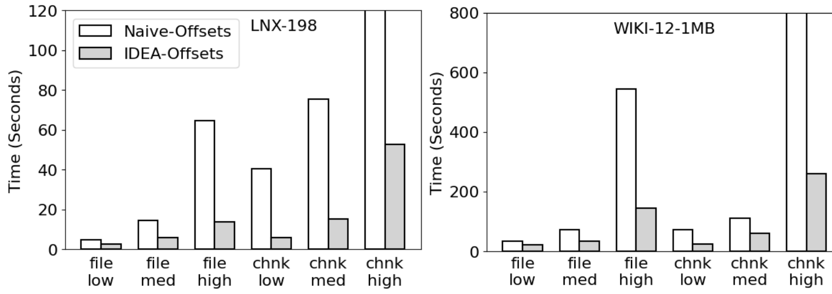


Figure 15: lookup times with 128 keywords of different dictionaries, in the LNX-198 (left) and WIKI-12-1MB (right) datasets, with offsets.

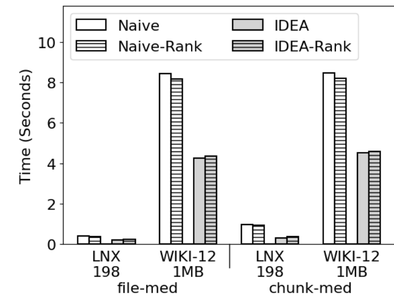


Figure 16: lookup times of 1 keyword of different dictionaries with TF-IDF ranking.

Dataset	LNX-198	Wiki-12-1MB
Naïve-rank	1.57H (15%)	36H (19%)
IDEA-rank	0.4H (18%)	13H (25%)

Table 5: Indexing times with ranking. The number in parentheses is the increase compared to the version without ranking.

and chunk-med dictionaries. The standard deviation was at most 0.4%. The lookup time of IDEA-rank is longer than that of IDEA by up to 25% (except a 50% increase in the "-low" dictionaries). This increase is due to the storage of the term offsets, which increases the term-to-chunk map. Interestingly, in Naïve, this increase in size triggered a segment split in Lucene, which results in slightly faster lookups. This inverse effect causes one anomaly: IDEA-rank is slower than Naïve-rank for words in the file-low dictionary and LNX-198. In all other cases, IDEA is much faster than Naïve.

8 Discussion and Open Challenges

Deduplication-aware indexing opens up several additional venues for improving search performance and applicability. The major advantage of our approach is its generality: it is orthogonal to the specific design details of both deduplication and indexing mechanisms. Deduplication-aware indexing can be integrated into any deduplication system that chunks incoming data streams. The design of the index itself relies on the basic search functionality of Lucene, and could use any other search engine. The chunk-to-file and file-to-path maps can be realized with any data structure or external database. Furthermore, the lookup in the two maps, term-to-chunk and chunk-to-file can be pipelined to reduce some of its overhead: looking up the unique chunks does not require that all of them are identified in advance.

IDEA can support file deletion similarly to existing index designs. The index must maintain the property that it returns all non-deleted files in the storage system that contain the query terms. This can be realized by marking each file as live or deleted, and returning only live files in the query result. A long series of file deletions, can, as in existing index designs, trigger garbage collection and an update of the term-to-chunk and chunk-to-file maps.

While our deduplication-aware indexing approach lends itself to many extensions and improvements, its dependency on white-space aware chunking might prevent it from being applicable when the system receives chunked data and does not perform chunking internally (such as in the case of existing deduplicating storage devices). When terms might be split between chunks, IDEA will have to process each chunk in the context of the chunks adjacent to it in each file.

A similar challenge is presented by files containing compressed text, such as `.pdf` or `.docx`. Their textual content can only be processed after the file is opened by a suitable application or converted by a dedicated tool. Thus, the individual chunks cannot be processed during offline index creation. Both challenges might be addressed by inline indexing, but will require adjusting the indexing process and data structures accordingly. We leave such extensions for future work.

Finally, the overhead of creating and storing an index might be prohibitively high, for deduplicated as well as non-deduplicated data. The choice between indexing and exhaustive search depends on the context of each specific system: its data type and the frequency and type of queries it is expected to serve. IDEA and IDEA-Direct introduce additional design choices between inline and offline indexing, and using HDD or SSD for external map structures.

9 Conclusions

Since most storage in large-scale systems is or will be deduplicated, standard storage functionality can be made more efficient by taking advantage of deduplicated state. In this paper, we presented the first design of a deduplication-aware term index. Our evaluation showed the advantages of this approach, as well as its flexibility in supporting advanced search functions.

Acknowledgments

We thank our shepherd, Vasily Tarasov, for his valuable feedback and suggestions. We thank Amnon Hanuhov and Nadav Elias for their help with the IDEA prototype, and Dafna Sheinvald for insightful discussions. This research was supported, in part, by the Israel Science Foundation (grant No. 807/20).

Artifact Appendix

Abstract

IDEA is a *deduplication-aware keyword index* which addresses the inherent challenges of indexing deduplicated data. We make IDEA's code, scripts, keywords, and configurations publicly available to allow the reproducability of our results and to facilitate further research of deduplication-aware indexing. This section describes the artifact that is made available with the publication of this paper.

Scope

The artifact includes code, tools, and instructions sufficient for reproducing the results presented in the paper. The instructions in the repository can be followed for generating result data equivalent to that used for Figures 7– 16. They can be used to verify the main claims of the paper:

- Indexing is faster with IDEA than with Naïve. This claim holds in setups with and without offsets and ranking, for inline as well as offline indexing.
- The Naïve index is larger than that of IDEA when the deduplication ratio and the number of files are high.
- When offsets are supported, the index size of Naïve is always larger than that of IDEA.
- Index size and indexing time increase when offsets or ranking are supported.
- Lookup times are faster in IDEA than in Naïve, except in several extreme cases.
- Offsets and ranking increase lookup times.

In addition to the reproducible environment, the repository contains instructions for modifying and recompiling the code.

Contents

The main content of the artifact is the source code of IDEA, which is a fork of Destor [46]. The artifact contains source and header files (`src/`), binary libraries of LucenePlusPlus [9] (`libs/`), and compilation scripts for building the IDEA executable. The same executable is also used to run the versions of the Naïve index. The repository also contains resources for reproducing the experiments described in this paper: instructions for downloading and creating the Linux and Wikipedia datasets (`dataset_details/`), the respective keywords (`keywords/`), relevant system configurations (`configs/`), and scripts for running the experiments (`scripts/`).

Hosting

Our artifact is hosted in GitHub and is available here: https://github.com/asaflevi0812/IDEA. The *main* branch is stable for installation and is up-to-date. To modify the code, either open the repository in an IDE in the host machine (e.g., over SSH), or fork the repository and use git to transfer changes between the virtual and host machines.

Requirements

Our prototype is based on Destor, which requires Ubuntu version 16.04. To be consistent with the evaluation setup described in this paper, follow the instructions in GitHub for creating the backup and index on HDD, with IDEA's external data structures on SSD. The required storage capacity depends on the dataset (see Table 1 for details). However, we reproduced our main results also in a server with HDD only, and on a server with Amazon AWS EBS SSD storage.

During the artifact evaluation process, our artifact was evaluated using an *M5.large* instance on AWS with 8 GB DRAM. The image name was *ubuntu-xenial-16.04-amd64-pro-server-20230912* and the storage was configured as GP3 100GB with 3000 IOPS.

References

[1] Amazon OpenSearch™. https://aws.amazon.com/what-is/opensearch/.

[2] Apache Lucene. https://lucene.apache.org/. Accessed: 2022-05-14.

[3] DedupSearch implementation. https://github.com/NadavElias/DedupSearch.

[4] Elasticsearch: The heart of the free and open Elastic Stack. //https://www.elastic.co/elasticsearch/.

[5] IBM Watson. https://www.ibm.com/watson.

[6] Indri. http://www.lemurproject.org/indri/.

[7] Linux Kernel Archives. https://mirrors.edge.kernel.org/pub/linux/kernel/.

[8] LMDB. http://www.lmdb.tech/doc/.

[9] LucenePlusPlus. https://github.com/luceneplusplus/LucenePlusPlus. Accessed: 2022-12-01.

[10] Meilisearch. https://www.meilisearch.com/.

[11] Oracle Berkeley DB. https://www.oracle.com/database/technologies/related/berkeleydb.html.

[12] RocksDB. http://rocksdb.org/.

[13] Solr. https://solr.apache.org/.

[14] TypeSense. https://typesense.org/.

[15] Wikimedia data dump torrents. https://meta.wikimedia.org/wiki/Data_dump_torrents.

[16] Wikimedia downloads. https://dumps.wikimedia.org/enwiki/.

[17] Microsoft search server 2010 expres. https://www.microsoft.com/en-us/download/details.aspx?id=18914, 2019.

[18] Fast and reliable schema-agnostic log analytics platform. https://www.uber.com/en-CA/blog/logging/, 2021.

[19] Commvault documentation pdf: Protect. access. comply. share. https://documentation.commvault.com/commvault/index.html, 2022.

[20] Deduplication: btrfs wiki. https://btrfs.wiki.kernel.org/index.php/Deduplication, 2022.

[21] Dell EMC Data Protection Search 19.6.1 deployment and administration guide. https://www.dell.com/support/home/en-il/product-support/product/data-protection-search/docs, 2022.

[22] Reducing logging cost by two orders of magnitude using CLP. https://www.uber.com/en-US/blog/reducing-logging-cost-by-two-orders-of-magnitude-using-clp, 2022.

[23] Searching 1.5tb/sec: Systems engineering before algorithms. https://www.dataset.com/blog/systems-engineering-before-algorithms/, 2022.

[24] Solaris ZFS administration guide: The dedup property. https://docs.oracle.com/cd/E19120-01/open.solaris/817-2271/gjhav/index.html, 2022.

[25] Veeam backup & replication 11: User guide for VMware vSphere. https://helpcenter.veeam.com/docs/backup/vsphere/overview.html?ver=110, 2022.

[26] Efficient search in netapp data storage solutions. https://intrafind.com/en/blog/efficient-search-in-netapp-data-storage-solutions, 2023.

[27] Windows search overview. https://learn.microsoft.com/en-us/windows/win32/search/-search-3x-wds-overview, 2024.

[28] Yamini Allu, Fred Douglis, Mahesh Kamat, Ramya Prabhakar, Philip Shilane, and Rahul Ugale. Can't we all get along? Redesigning protection storage for modern workloads. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[29] Alexandr Andoni. Nearest neighbor search: the old, the new, and the impossible. Ph.d. thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2009.

[30] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS 09)*, 2009.

[31] Renata Borovica-Gajić, Raja Appuswamy, and Anastasia Ailamaki. Cheap data analytics using cold storage devices. *Proceedings of the VLDB Endowment*, 9(12):1029–1040, 2016.

[32] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[33] Zhichao Cao, Hao Wen, Fenggang Wu, and David H.C. Du. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018.

[34] Feng Chen, Tian Luo, and Xiaodong Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, 2011.

[35] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

[36] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation (OSDI 04)*, 2004.

[37] Wei Dong, Fred Douglis, Kai Li, Hugo Patterson, Sazzala Reddy, and Philip Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, 2011.

[38] Fred Douglis, Deepti Bhardwaj, Hangwei Qian, and Philip Shilane. Content-aware load balancing for distributed backup. In *25th International Conference on Large Installation System Administration (LISA 11)*, 2011.

[39] Fred Douglis, Abhinav Duggal, Philip Shilane, Tony Wong, Shiqin Yan, and Fabiano Botelho. The logic of physical garbage collection in deduplicating storage. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017.

[40] Abhinav Duggal, Fani Jenkins, Philip Shilane, Ramprasad Chinthekindi, Ritesh Shah, and Mahesh Kamat. Data Domain Cloud Tier: Backup here, backup there, deduplicated everywhere! In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[41] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary data deduplication—large scale study and system design. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012.

[42] Nadav Elias, Philip Shilane, Sarai Sheinvald, and Gala Yadgar. DedupSearch: Two-Phase deduplication aware keyword search. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, 2022.

[43] EMC Corporation. *Introduction To The EMC XtremIO Storage Array (Ver. 4.0)*, rev. 08 edition, April 2015.

[44] Jingxin Feng and Jiri Schindler. A deduplication study for host-side caches in virtualized data center environments. In *29th IEEE Symposium on Mass Storage Systems and Technologies (MSST 13)*, 2013.

[45] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[46] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. Design tradeoffs for data deduplication performance in backup workloads. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015.

[47] Clinton Gormley and Zachary Tong. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine*. O'Reilly Media, Inc., USA, 2015.

[48] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. Retrieval augmented language model pre-training. In *International Conference on Machine Learning*, pages 3929–3938. PMLR, 2020.

[49] Danny Harnik, Moshik Hershcovitch, Yosef Shatsky, Amir Epstein, and Ronen Kat. Sketching volume capacities in deduplicated storage. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019.

[50] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[51] Roei Kisous, Ariel Kolikant, Abhinav Duggal, Sarai Sheinvald, and Gala Yadgar. The what, the from, and the to: The migration games in deduplicated systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, 2022.

[52] Naama Kraus, David Carmel, and Idit Keidar. Fishing in the stream: similarity search over endless data. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 964–969. IEEE, 2017.

[53] Geoff Kuenning. How does a computer virus scan work? *Scientific American*, January 2002.

[54] Mike Lewis, Marjan Ghazvininejad, Gargi Ghosh, Armen Aghajanyan, Sida Wang, and Luke Zettlemoyer. Pretraining via paraphrasing. *Advances in Neural Information Processing Systems*, 33:18470–18481, 2020.

[55] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[56] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. CacheDedup: In-line deduplication for flash caching. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[57] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013.

[58] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, USA, 2008.

[59] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, 2011.

[60] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems (TOIS)*, 14(4):349–379, 1996.

[61] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-*

Throughput Sequencing. Cambridge University Press, 2015.

[62] Aviv Nachman, Gala Yadgar, and Sarai Sheinvald. GoSeed: Generating an optimal seeding plan for deduplicated storage. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.

[63] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David H. C. Du. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *2011 IEEE International Conference on High Performance Computing and Communications (HPCC 11)*, 2011.

[64] Enno Ohlebusch and Giuseppe Ottaviano. 3.16 bioinformatics algorithms: Sequence analysis, genome rearrangements, and phylogenetic reconstruction. *Indexes and Computation over Compressed Structured Data*, page 33, 2013.

[65] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proc. VLDB Endow.*, 8(12):1816–1827, Aug 2015.

[66] Juan Ramos. Using TF-IDF to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 29–48, 2003.

[67] Martin H Redish. Electronic discovery and the litigation matrix. *Duke Law Journal*, 51:561, 2001.

[68] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of inverted indexes for fast query evaluation. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '02, 2002.

[69] Philip Shilane, Ravi Chitloor, and Uday Kiran Jonnala. 99 deduplication problems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[70] Fabrizio Silvestri. *Mining Query Logs: Turning Search Usage Data into Knowledge*, volume 4. Now Publishers Inc., Hanover, MA, USA, Jan 2010.

[71] Malcolm Slaney and Michael Casey. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *IEEE Signal Processing Magazine*, 25(2):128–131, 2008.

[72] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, 2012.

[73] Ja-Hwung Su, Yu-Ting Huang, Hsin-Ho Yeh, and Vincent S Tseng. Effective content-based video retrieval using pattern-indexing and matching techniques. *Expert Systems with Applications*, 37(7):5068–5085, 2010.

[74] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proceedings of the VLDB Endowment*, 6:1930–1941, 09 2013.

[75] Ke Tao, Fabian Abel, Claudia Hauff, and Geert-Jan Houben. Twinder: A search engine for twitter streams. In Marco Brambilla, Takehiro Tokuda, and Robert Tolksdorf, editors, *Web Engineering*, pages 153–168, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[76] Jau-Hwang Wang, Peter S Deng, Yi-Shen Fan, Li-Jing Jaw, and Yu-Ching Liu. Virus detection using data mining techinques. In *IEEE 37th Annual 2003 International Carnahan Conference on Security Technology.* IEEE, 2003.

[77] Kenneth J Withers. Computer-based discovery in federal civil litigation. *Fed. Cts. Law Rev.*, 1:65, 2006.

[78] Ian H Witten, Ian H Witten, Alistair Moffat, Timothy C Bell, Timothy C Bell, Ed Fox, and Timothy C Bell. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.

[79] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[80] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, 2008.

MiDAS: Minimizing Write Amplification in Log-Structured Systems through Adaptive Group Number and Size Configuration

Seonggyun Oh* DGIST
Jeeyun Kim* DGIST
Soyoung Han DGIST
Jaeho Kim Gyeongsang National University
Sungjin Lee DGIST
Sam H. Noh Virginia Tech

Abstract

Log-structured systems are widely used in various applications because of its high write throughput. However, high garbage collection (GC) cost is widely regarded as the primary obstacle for its wider adoption. There have been numerous attempts to alleviate GC overhead, but with ad-hoc designs. This paper introduces MiDAS that minimizes GC overhead in a systematic and analytic manner. It employs a chain-like structure of multiple groups, automatically segregating data blocks by age. It employs analytical models, Update Interval Distribution (UID) and Markov-Chain-based Analytical Model (MCAM), to dynamically adjust the number of groups as well as their sizes according to the workload I/O patterns, thereby minimizing the movement of data blocks. Furthermore, MiDAS isolates hot blocks into a dedicated *HOT* group, where the size of *HOT* is dynamically adjusted according to the workload to minimize overall WAF. Our experiments using simulations and a proof-of-concept prototype for flash-based SSDs show that MiDAS outperforms state-of-the-art GC techniques, offering 25% lower WAF and 54% higher throughput, while consuming less memory and CPU cycles.

1 Introduction

Log-structured systems are widely used in various applications such as key-value stores (*e.g.,* LSM-trees [36, 40]), file systems (*e.g.,* F2FS [28]), and storage firmware (*e.g.,* FTLs [18, 22, 24, 31]). Log-structured systems not only provide high write throughput with fairly good latency, but are also well-suited for emerging storage media that only supports append-only writes such as NAND flash-based devices [3, 9], ZNS [6, 34, 45], and SMR [1, 2, 17] drives.

Despite such benefits, the high garbage collection (GC) cost of log-structured systems is considered its major impediment. Log-structured systems divide the storage space into fixed-size segments, each several MiB in size, while the segments themselves comprise 4KiB data blocks. A segment is the unit of space allocation and GC, and a 4KiB block is the unit of reading and writing data. Log-structured systems append new versions of data blocks to segments, leaving old ones as garbage that must be cleaned up through GC later. During GC, a victim segment with garbage blocks is identified, valid (or live) blocks are copied to another segment, and finally the new freed victim segment is returned for future writes. Relocating live blocks causes numerous extra reads and writes. A common metric to measure the impact of extra writes during GC is the *write amplification factor* (WAF), which is the ratio of the total number of blocks written to storage to the number of blocks written by the user.

Many studies have been conducted to alleviate the overhead caused by GC. These studies try to reduce WAF by employing two main techniques: victim selection [15, 23, 38] and data placement [10, 11, 27, 33, 33, 37, 42, 44, 49–51]. Despite these many efforts, existing techniques often fail to minimize WAF because of the following two limitations. The first is inaccurate prediction of block lifespan, that is, distinguishing hot and cold blocks. Hot blocks (frequently updated blocks) have short lifespan while cold blocks (infrequently updated blocks) have long lifespan. While the notion of hot and cold is well accepted, the boundary between hot and cold is relative according to workload and typically changes over time. Existing techniques cannot efficiently define such a boundary, thereby making inaccurate classification of data blocks. This results in data blocks with varying lifespans being mixed up in the same segment causing many live block copies during GC. The second is inefficient partitioning of storage space. To group blocks with similar lifespan together, existing techniques maintain groups of segments and segregate data blocks with similar lifespan to a designated group. Current state-of-the-art techniques typically work with 2–8 groups. Unfortunately, the number of groups and their sizes are decided in an ad-hoc manner resulting in suboptimal WAF.

In this paper, we propose MiDAS, a **M**igration-based **D**ata placement technique with **A**daptive group number and **S**ize configuration for log-structured systems. MiDAS employs a chain-like structure comprising multiple groups, with each group G_i linked to the subsequent group G_{i+1}. Incoming data blocks are initially written to the first group G_1 and thereafter, only valid blocks from one group G_i are moved to the next group G_{i+1} automatically segregating data blocks by age.

*These authors equally contributed to this work.

MiDAS optimizes the number of groups and their sizes according to the characteristics of the workload so that the number of valid block copies between segments is minimized. This is done by making use of analytical models, Update Interval Distribution (UID) and Markov-Chain-based Analytical Model (MCAM). By monitoring long-term trends of block updates, UID tells us how many blocks in a group remain valid and move to the next group. By leveraging the chained organization of groups in MiDAS, MCAM accurately predicts the WAF value given a specific group configuration. Using UID and MCAM, MiDAS explores a wide range of group configurations and finds one that minimizes WAF.

To further reduce WAF, MiDAS also isolates hot blocks in a group called HOT. To identify hot blocks, MiDAS does not rely on simple heuristics. Instead, MiDAS sends selected blocks that are soon to be invalidated to the HOT group, which is dynamically adjusted according to changing workloads in balance with other groups to minimize overall WAF.

While MiDAS is designed for log-structured systems, this paper specifically focuses on data placement for flash-based SSDs for evaluation. Accordingly, we have implemented MiDAS on the FTL, and all experiments are conducted at the SSD level. To understand the effectiveness of MiDAS, we conduct a simulation study using I/O traces collected from various benchmarks and real-world systems. We compare MiDAS to four state-of-the-art GC techniques: CAT [10], AutoStream [51], MiDA [37], and SepBIT [44]. Our results show that MiDAS can provide 25% lower WAF compared to the other techniques, on average. We also implement a proof-of-concept prototype of MiDAS in an SSD controller and confirm that MiDAS not only provides lower WAF and higher throughput, but exhibits better memory efficiency and consumes fewer CPU cycles than SepBIT, which is the best-performing state-of-the-art (SOTA) technique. We also include a discussion on the applicability of MiDAS to other log-structured systems.

2 Background and Related Work

2.1 Victim Selection Policies

A victim selection policy decides a victim segment with the goal of minimizing the number of live block copies during GC. Three commonly used policies are (*i*) *FIFO* [15,38], (*ii*) *Greedy* [38,47], and (*iii*) *Cost-Benefit* [10,23,38].

FIFO chooses the oldest segment as a victim. FIFO is simple to implement, but often misses opportunities to select better segments with fewer live blocks as victims.

Greedy selects the segment with the lowest utilization u (the fraction of blocks still live) as u determines the number of valid block copies. It reclaims the largest fraction of the segment space 1-u after GC. Greedy, however, often selects segments containing hot blocks, which need not be copied during GC as they will soon be invalidated [15,38,47].

Cost-Benefit (CB) aims to minimize GC cost by considering both the utilization and age of the segment [38] trying to avoid unnecessary copies of hot blocks. CB calculates scores for individual segments and selects one with the minimum score. A commonly used score is $\frac{u}{age\times(1-u)}$, where age represents how long the segment has been alive since its creation.

CB exhibits lower WAF than FIFO and Greedy. Despite its higher complexity, CB is widely adopted for low GC cost. The effectiveness of the victim selection policy, however, is highly correlated with the data placement policy being used.

2.2 Data Placement Policies

Various data placement policies have been proposed to further reduce WAF. The fundamental idea behind data placement is to group together data blocks with similar *invalidation time*, that is, the blocks that are likely to be invalidated at a similar time. Before writing a user data block (a user-written block) or relocating a live block from a victim segment to another, the data placement policy estimates the expected invalidation time of the block and then assigns it to an appropriate segment that is holding blocks with similar invalidation time. Then, as the blocks in the segment are all invalidated at similar times, this assists in generating dead segments, which contain only invalid blocks. Dead segments do not require any live block copies, so WAF can be significantly reduced.

The key here is in accurately estimating the invalidation time of a block. While many strategies have been suggested [21,25,33,44,49,50], one or a combination of the following three attributes of a block is commonly used: (*i*) *update frequency* [10,11,27,33,42,51], (*ii*) *latest update interval* [44,50], and (*iii*) *age of the block* [33,37,44,49].

2.3 Review of Prior Techniques

Here, we present four SOTA GC techniques, CAT [10], AutoStream [51], MiDA [37], and SepBIT [44], focusing on the data placement method used. CAT [10] categorizes a data block into two types, hot and cold, based on its update frequency and assigns it to either a hot or cold segment group. It dynamically changes the sizes of groups by moving live blocks over groups during GC.

AutoStream [51] designed for Multi-Streamed SSDs [16] attempts to finely categorize data blocks at the host level with support from the storage device. AutoStream counts the number of updates of data blocks on the host side and classifies them based on update frequency. Then, it sends data blocks with group IDs determined by their update frequencies to the SSD. Owing to the limit of being designed in the confines of an SSD, the number of groups is usually set to five [21].

MiDA [37] utilizes the age of a block for data placement. It creates a chain of segment groups, each of which is connected to a neighboring group. Incoming blocks are first written to Group 1, the head of the chain, with an age of 0. If it remains valid until being selected as a victim for GC, it is moved to the next group, Group 2, with an age of 1, and so on. In this way, it clusters data blocks with similar ages in the same group. There is no specific limit on the number of groups, but MiDA maintains up to eight groups by default.

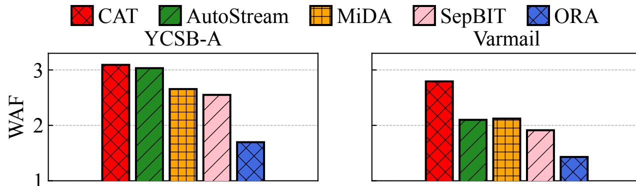


Fig. 1. WAF of ORA and existing SOTA techniques

SepBIT [44] employs both the latest update interval and age of a block. For a newly written block, it estimates the invalidation time based on its latest update interval – the timespan since the block was last written – and assigns the block to hot or cold groups. When relocating a live block during GC, SepBIT measures the age of the block and sends it to an appropriate GC group. The total number of groups, including hot, cold, and GC groups, is six. SepBIT uses a threshold-based heuristic to decide a target group where data blocks are assigned.

In summary, first, most GC techniques based on invalidation times usually group segments into 2–8 groups, and this number of groups is decided without any justification. We are aware of some prior studies that address this by dynamically changing the number of GC groups [42, 48]. For instance, Multilog estimates the update frequency of a block by employing both the LRU and Oracle algorithms [42]. If a block's update frequency falls below the average, a colder group is created, and the block is demoted to this group. Also, a comprehensive analysis of the best number of GC groups across various workloads is provided by Yadgar et al. [48]. However, these studies did not consider how the number of groups should change as the workload dynamically changes. Second, the size of the group, though dynamic in a limited way, is determined without considering what size is most appropriate to accommodate the incoming blocks destined for the group. We are not aware of any prior work that tackles this issue.

3 Motivation: Current GC Techniques

In this section, we analyze the limitations of current SOTA GC techniques through quantitative observations, which serve as motivation of our work MiDAS.

3.1 Experimental Setup

To evaluate the effect of current SOTA GC techniques, we implement the techniques within the FTL of flash-based SSDs, and make use of two benchmarks, YCSB-A [13] that runs on MySQL and Varmail of the Filebench benchmark [43]. The number of 4KiB blocks written by YCSB-A and Varmail are 4.4 billion (16.4 TiB) and 4 billion (14.9 TiB), respectively. To repeat the experiments under the same environment, the I/O traces of these benchmarks are collected and fed to a trace-driven simulator that implements the victim selection and data placement policies of the various GC techniques. The simulator models a 128GiB storage space with 64MiB segments. The experimental setup is detailed in §5.

To objectively evaluate the performance of the existing techniques, we compare them with an oracle algorithm (ORA) that minimizes WAF through offline analysis of the collected traces. Through this analysis, the invalidation time of every block is obtained, which is then used to assign blocks with similar invalidation times to the same segment group. Note, however, that deciding the optimal number of segment groups and group sizes is an NP-hard problem [29]. Thus, we perform k-means clustering [20] over the traces to find the best number of groups that accommodates data blocks with similar invalidation time. We also empirically decide the size of individual groups such that WAF is lowest. Note that ORA does not adjust the group size during victim selection because their sizes are decided a priori.

Table 1: Ranges of invalidation times for C_1–C_6 in ORA

	C_1	C_2	C_3	C_4	C_5	C_6
Range	<250K	250K–5M	5M–14M	14M–28M	28M–62M	>62M

The four SOTA techniques reviewed in §2.3 are compared against ORA. According to their original design, the number of segment groups is set to 2, 5, 8, and 6 for CAT, AutoStream, MiDA, and SepBIT, respectively. The victim selection policy is CB as it provides the best performance.

Fig. 1 shows the WAF results. Note that when reporting WAF values, throughout the paper, we consistently start from base 1 for the y-axis. ORA is effective with WAF being closest to 1.0 for both YCSB-A and Varmail. CAT shows the worst WAF. AutoStream, MiDA, and SepBIT, which maintain multiple groups for data placement and use more sophisticated invalidation time prediction, exhibit lower WAF. However, we still see a large gap between the existing techniques and ORA. In the following, we conduct a series of experiments to analyze where this discrepancy is coming from.

3.2 Analysis based on ORA

Accuracy of invalidation time prediction: We first evaluate how the accuracy of invalidation time prediction varies per prediction approach. Invalidation time is defined to be the number of user-written blocks, in 4KiB units, which are written between the time the block of interest is written to and the time it becomes invalid. Blocks with shorter invalidation time generally mean they are hotter.

To objectively evaluate prediction accuracy, we utilize the classifications of blocks by ORA that accurately groups data blocks depending on their actual invalidation times through offline analysis. The analysis results in ORA dividing the data blocks into six categories, C_1, C_2, ..., C_6, depending on their hotness. C_1 represents the hottest blocks (invalidation time < 250K), C6 represents the coldest (invalidation time > 62M), and the rest are in between. Table 1 lists the ranges of invalidation times for C_1, C_2,..., C_6.

Fig. 2 illustrates the accuracy of predicting invalidation times for the YCSB-A[1] workload using three different approaches: the latest update interval (employed in SepBIT),

[1]The results trend and discussions are similar for Varmail and thus, are not presented in the interest of space.

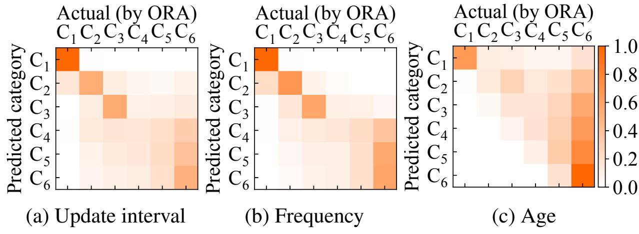


Fig. 2. Accuracy of block invalidation times for YCSB-A according to prediction techniques

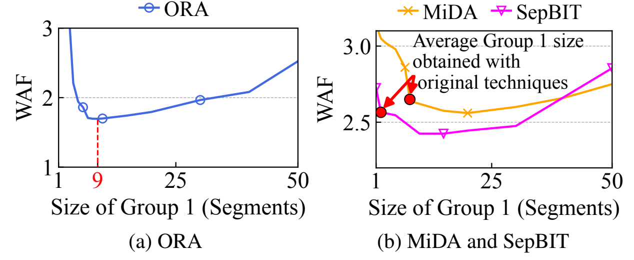


Fig. 3. Impact of group size on WAF for YCSB-A

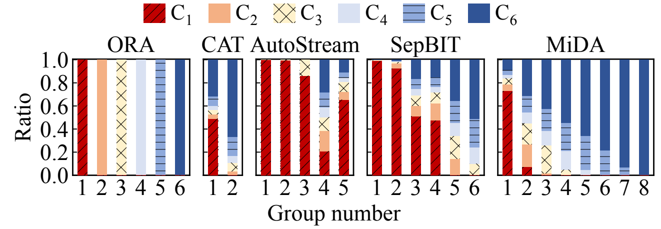


Fig. 4. Distribution of blocks over groups for YCSB-A

Table 2: Size of each group per technique for YCBS-A (unit: segment): fixed values for ORA, while the rest are averages through workload processing

Group number	1	2	3	4	5	6	7	8
ORA	9	17	35	45	133	1,809	-	-
CAT	184	1,864	-	-	-	-	-	-
AutoStream	2	3	2	1,336	705	-	-	-
SepBIT	2	1	17	2	85	1,941	-	-
MiDA	7	79	95	126	128	117	98	1,398

update frequency (utilized in CAT and AutoStream), and the age of a block (employed in SepBIT and MiDA). In Fig. 2, we visualize the prediction accuracy of these techniques in a heatmap, comparing them to ORA. The x-axis represents the category of blocks that are decided by ORA through offline analysis. The y-axis represents the category of blocks that are predicted by each approach, also in an offline manner. More specifically, we first categorize each block as C_i based on ORA. Then, we categorize the blocks again, this time using the specific approach. For example, say there is a block A that is categorized as C_2 with ORA. Then, with the latest update interval approach, say, we observe that block A is updated at time 200K, which is in the C_1 range. Then, this block A will be a miscount that reduces the accuracy of the (C_2, C_2) zone of Fig. 2(a) and that contributes to the (C_2, C_1) zone. Thus, the intensity of the diagonal zones shows how accurate each approach is relative to ORA, while the non-diagonal zones show how much they are contributing to the inaccuracy. Ideally, if the predictions of each approach were perfect, we would only observe dark diagonal zones.

From Fig. 2, we find that the existing approaches show higher accuracy on different categories of block hotness. For latest update interval and update frequency, the prediction accuracy of hot blocks (*e.g.,* C_1), that is, those with short invalidation times, is relatively high. However, for cold blocks (*e.g.,* C_6), the prediction is much less accurate. We observe, for example, from Fig. 2(b), that blocks that actually belong to C_6 are incorrectly categorized as other C_is, even including C_2. Conversely, the age-based technique, Fig. 2(c), shows lower accuracy than the other two for hot blocks, but it excels in identifying the coldest blocks (C_6). We do see, however, that many coldest blocks are being mispredicted as other blocks ($C_1, \ldots, C_5$). This is likely due to the fact that many of coldest blocks are still in transit and moving towards the coldest block at the end of the experiments.

Effect of group number and size: Once we have assessed the individual block's hotness, they need to be grouped together according to their hotness. ORA, which accurately assesses hotness, through offline analysis, came up with six groups of sizes as shown in the 'ORA' column of Table 2. Fig. 3(a), where the x-axis is the size of Group 1 and the y-axis is WAF, shows how WAF changes as the ORA Group 1 size is varied. It shows that ORA chose the appropriate Group 1 size and that even with an accurate hotness assessment, incorrectly setting the group size can amplify WAF. As analysis determined that six groups show the best WAF for ORA, altering the number of groups will show similar WAF amplification.

3.3 Analysis of SOTA Techniques

As discussed, inaccurate data placement comes from two sources, inaccurate hotness predictions and inaccurate group configurations, that is, group number and size. We now attempt to quantify these issues for the four SOTA techniques.

Fig. 4 illustrates the distribution of data blocks over segment groups for the SOTA techniques. We observe that results for AutoStream based on the update frequency and SepBIT based on the update interval coincide well with the findings shown in Fig. 2, with Group 1 comprising mostly of C_1, the hot blocks. However, as shown in Table 2, the sizes for Groups 1 to 3 for these techniques are considerably smaller than those of ORA, leading to many of the hot blocks overflowing to other colder groups. We observe that for Autostream, with only five groups, the hot blocks are scattered among all the groups. Similarly, the results of the age-based MiDA technique, which generates eight groups, also coincide well with the findings shown in Fig. 2, with the coldest blocks fully filling Group 8 and forming the majority of Groups 5–7, while Group 1 shows some of its space being occupied by colder blocks. We also observe that CAT cannot perform well

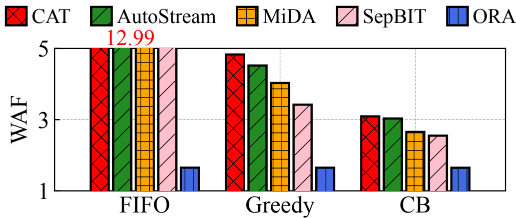


Fig. 5. WAFs of each technique by victim selection (FIFO WAF for all techniques excluding ORA is 12.99)

due to being fixed to two groups resulting in an intermix of hot and cold blocks.

To show the effect of group size on WAF, we manually control the size of Group 1 while running YCSB-A for MiDA and SepBIT. The results are shown in Fig. 3(b). Note that in our setup, as well as in typical systems, the total number of segments is fixed. Thus, as Group 1 size increases, the other groups will become smaller. The red dots in the figure show the average size of Group 1 and their original WAF values. We observe that by increasing the group size to only about 20 segments, both MiDA and SepBIT can reduce WAF by about 5.5% and 7.7%, respectively. However, further increasing the group size results in higher WAF due to the size reduction in subsequent groups. Based on these observations, the challenge becomes how to determine the number of groups to maintain and what the sizes of these groups should be such that WAF may be minimized.

Impact of victim selection: Lastly, we consider the effect of victim selection on WAF. To this end, we measure the WAF values of the five techniques with three victim selection policies: FIFO, Greedy, and CB.

Fig. 5 shows the results, from which we make two observations. First, each technique exhibits the lowest WAF when employing CB, which is a predictable outcome. This is because CB provides sufficient time for hot blocks to become invalid. Second, ORA exhibits almost the same WAF values, regardless of which victim selection policy is used. Even with FIFO, which is the simplest and where other techniques suffer, ORA can achieve low WAF. This is an interesting, yet expected result. If data blocks are perfectly distributed over different segment groups according to exact invalidation times and the group sizes are set sufficiently large, the oldest segment in the group will have the least number of valid blocks that will not be invalidated for a long time, eventually trickling down to the last group. As a result, FIFO, Greedy, and CB all behave similarly, showing almost the same WAF.

3.4 Lessons Learned: A Summary

From the results above, we make the following three key observations. *Observation #1.* Current SOTA techniques are inaccurate in predicting hotness of data. However, latest update interval and update frequency based prediction approaches tend to predict hot data relatively well, while, in contrast, age-based prediction approaches tend to predict cold data relatively well. A mix of these methods should help improve

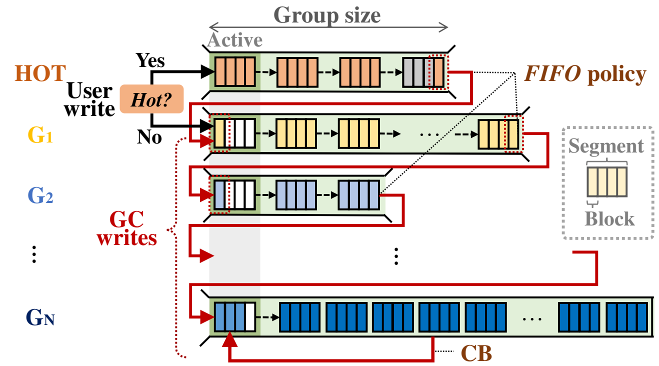


Fig. 6. Overview of MiDAS

overall predictions. *Observation #2.* The number of segment groups and their sizes have a critical impact on GC efficiency. Effort should be put into finding group number and size values that minimize WAF. *Observation #3.* FIFO is equally efficient as any elaborate policy when group sizes are properly set. With a correct data placement framework, FIFO should suffice as a victim selection policy.

4 Design of MiDAS

In this section, we present MiDAS, a technique that automatically determines the number of segment groups and their sizes to minimize WAF according to the given workload. As §3.2 illustrates, MiDAS separates the cold blocks using an age-based policy as does MiDA and separates hot blocks using update intervals as does SepBIT. In the following, we first give a high level overview of MiDAS, focusing on the relations among the key components such as the *HOT* group, UID, and MCAM. Then, in the subsequent subsections, each of these components are described in detail along with how these components interact.

4.1 Overview of MiDAS

Fig. 6 depicts the overall organization of MiDAS with $N+1$ segment groups, one *HOT* group and N cold groups, G_1, G_2, ..., G_N. From *HOT* to G_N, each segment group is linked to its next group, which creates a chain of segment groups. Upon arrival of a user-written block, the block is determined to be a hot block or not (described in §4.2). Hot blocks are directed to the *HOT* group, while others are sent to G_1, bypassing *HOT*. Every segment group, including *HOT*, has a designated size. Once the group becomes full with data blocks, a victim segment is selected from that group. Then, live blocks from this victim are migrated to the next group, G_1 for *HOT* and G_{i+1} for G_i. The freed segment is returned to the original group. The last group, G_N, does not have a next group. Thus, valid blocks from the victim are sent to G_N again, and the free segment is returned to G_N as well.

Based on the observation in §3.2, the update interval can be used as a useful means of detecting hot blocks with short invalidation times. MiDAS segregates hot blocks into the separate *HOT* group based on their update intervals. To prevent

cold blocks from being incorrectly categorized and being mixed with hot blocks in HOT (as seen in CAT), MiDAS sends only data blocks with short update intervals to HOT. At the same time, to prevent hot blocks from being sent to cold segments owing to limited HOT group space (as seen in SepBIT), MiDAS dynamically adjusts HOT to have sufficient space to accommodate the identified hot blocks.

As we have also learned from §3.2, the age-based method is effective in separating cold blocks. Therefore, MiDAS segregates data blocks with the same age in the same group, sending older blocks to the next group. Here, age is defined to be the migration count from one group to another, as is done in MiDA. All data blocks in G_i thus have the same age of $i-1$. One exception is the last group G_N, where data blocks come from G_{N-1} and from itself, that is, G_N. The ages of blocks in G_N are greater than $N-2$.

The most crucial issue in designing MiDAS is to decide the number of groups and their sizes, so that the number of valid block copies between groups is to be minimized. To make accurate decisions, MiDAS monitors long-term behaviors of block updates and creates an Update Interval Distribution (UID) model. Given a segment group with a specific size, UID tells us how many blocks in the group stay alive and move to the next group. By leveraging the chained organization of segment groups, MiDAS employs a Markov-Chain-based Analytical Model (MCAM) that accurately predicts WAF for a given group configuration. By integrating UID and MCAM, we can explore a range of group configurations, which enables us to determine the most effective combination of the number of groups and group sizes that minimizes overall WAF.

If a proper group configuration is chosen by UID and MCAM, segment groups would have sufficiently large space so that blocks are invalidated prior to eviction. This allows MiDAS to manage each segment group as a FIFO queue and to use the simple FIFO victim selection policy.

If I/O patterns of the workload are irregular and change significantly over time, our models, UID and MCAM, which rely on past history to forecast future behavior, may not make accurate decisions. Then, MiDAS simply falls back to the basic MiDA technique.

4.2 Hot Block Separation

Prior data placement techniques take various approaches to define a boundary between hot and cold. MiDA simply segregates hot from cold blocks by sending old blocks to the next group in the chain. CAT and AutoStream explicitly define a hot-cold boundary based on update frequency (*i.e.,* update counts), but with disappointing results.

SepBIT uses a more advanced approach to define a hot-cold boundary. It internally maintains a queue and pushes block numbers of every user-written block into the queue. Then, user-written blocks referenced again within the queue are sent to a designated hot group. The length of the queue is set to the average resident time, which is the time user-written blocks remain in the hot group before being removed. The queue length depends on the hot group size, which is adjusted by CB. This is an interesting and novel approach where it tries to segregate hot blocks from the rest of the blocks by adjusting the queue length based on the lifetime of the hot blocks, that is, blocks residing in the hot group.

Unfortunately, SepBIT tends to misbehave owing to how its hot group size is decided. For example, let us assume that SepBIT accurately segregates hot blocks in the hot group. Then, many dead blocks are generated from the hot group and are quickly removed due to using CB. This reduces the average resident time, which in turn shrinks the queue size. This results in only a few hot blocks being sent to the hot group, even though many more hot blocks may exist. Conversely, if many cold blocks are mistakenly sent to the hot group, the length of the queue is likely to grow because it takes longer to evict blocks from the hot group. As a result, SepBIT may assign even more cold blocks to the hot group.

MiDAS tackles the limitations that the prior techniques have by taking two unique approaches: (*i*) tight admission control to the HOT group and (*ii*) dynamic size adjustment of the HOT group. The first, tight admission control, is similar to the approach of SepBIT, but MiDAS is more conservative when deciding a block to be hot. Similar to how SepBIT promotes blocks from the queue to the hot group, MiDAS promotes, from G_1, data blocks that are soon to be invalidated to the HOT group. Similar to how SepBIT maintains the average resident time to set the queue length, MiDAS maintains the same value and refers it as the threshold time. The difference, though, is that this threshold time is used to decide if the G_1 to HOT promotion should occur or not, instead of, for setting the queue length. Specifically, MiDAS compares the update interval of the block to the threshold time, and only blocks that are updated three times[2] within the threshold time are confirmed and promoted to HOT. Note that the choice to use the update interval to identify hot blocks is based on §3.2.

The second unique approach of MiDAS is that the size of HOT adapts to the workload. As we have seen in §3, as hotness is a relative notion, segregating hot blocks from the rest is not a simple matter, which is compounded by the difficulty of setting the size of the group that will hold the hot blocks. In MiDAS, the size of HOT is determined in conjunction with the rest of the group such that the overall WAF is minimized. The key technical issue, then, is how to estimate the impact of the group size adjustment, including that of HOT, on overall WAF, without reorganizing the actual group sizes, which is costly. MiDAS can accurately predict expected WAF using MCAM and UID as explained below.

4.3 Prediction of WAF using MCAM

We now explain MCAM, a Markov-Chain-based Analytical Model, to predict the WAF value of a given group configuration. Note that Bux and Iliadis calculate WAF using a Markov

[2] A 2-bit counter is used for this, whose overhead is analyzed in §5.1.

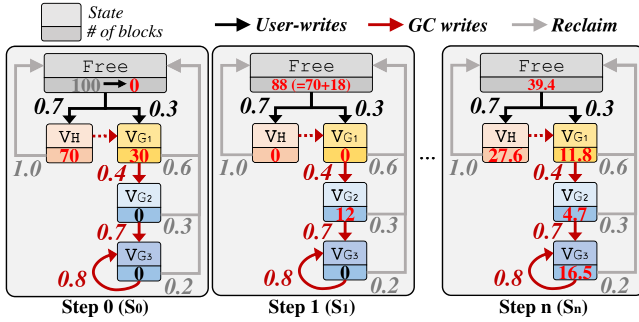


Fig. 7. Diagram of MCAM and WAF prediction process using MCAM as states transition

Chain for uniform workloads [8]. However, this prior work does not address multiple GC group scenarios needed for MiDAS, which, as depicted in Fig. 6, forms a chain of segment groups where live data blocks migrate between adjacent groups. Thus, we design MCAM to predict WAF across various group configurations, applicable to any workload pattern. If the input workload is in a steady state, MCAM is able to predict the value of WAF accurately by simulating the flow of data blocks moving over segment groups.

MCAM consists of states that blocks can be in and the transition probabilities between those states. Blocks can be in one of two states: valid (V) and free (Free). The valid state is categorized into finer states, V_H and V_{G_i}, for $i = 1, \ldots, N$, according to which group (*i.e., HOT* or G_i) the block is valid in. Hereafter, we use the index i to always represent the range $1, \ldots, N$ unless otherwise stated. Free indicates the state of the block that is invalidated, reclaimed, and ready to use.

We now discuss transitions between states and, for clarity, refer the reader to the leftmost figure in Fig. 7 as an example with four groups (*HOT*, G_1, G_2, and G_3). For transitions from V_H to V_{G_1} and from V_{G_i} to $V_{G_{i+1}}$, where $V_{G_{N+1}} = V_{G_N}$ (which means that V_{G_N} transitions into itself), live blocks in the victim segment chosen for GC are moved from the source to the destination. The rest of the blocks are invalidated and then become free, which forms the transitions from V_{G_i} to Free. Suppose that a fraction of valid blocks in the victim segment of G_1 is 0.4, on average. Then, the transition probability, which is the probability that blocks in one state move to another state, from V_{G_1} to V_{G_2} is 0.4 and the transition probability from V_{G_1} to Free is, naturally, 0.6. In MiDAS, user-written blocks are stored in free blocks and then assigned to either *HOT* or G_1. Thus, two transitions, from Free to V_H (0.7 in Fig. 7) and from Free to V_{G_1} (0.3 in Fig. 7), represent the movement of user-written blocks to *HOT* and G_1, respectively. The sum of the two transition probabilities is always 1.0. All the blocks destined for *HOT* are expected to be invalidated before eviction, and thus, the transition probability from V_H to Free is assumed to be 1.0 at all times.

Let us now discuss how WAF is predicted with MCAM using Fig. 7. We denote the progress of states as step S_k. For the moment, assume that the transition probabilities are given as in the figure. How these are obtained will be discussed in §4.4. Let us also assume that, at the initial step, S_0, we have 100 user-written blocks come in and the transition probabilities to *HOT* and G_1 are 0.7 and 0.3, respectively. Thus, we have 70 and 30 blocks in *HOT* and G_1, respectively. At the next step, S_1, 12 blocks in G_1 are moved to G_2, while the other 18 blocks are moved to Free by the transition probabilities from V_{G_1} to V_{G_2} (0.4) and from V_{G_1} to Free (0.6). No blocks in *HOT* move to G_1; instead, all blocks (70 blocks) move to Free as we expect all blocks in *HOT* to be invalidated.

This transition to the next step is repeated in similar manner until the number of blocks in each group converges, whose condition is met when the number of blocks in each state no longer changes. (Note that this process has been shown to converge [7].) S_n of Fig. 7 shows an example of how the converged results would look like. Then, WAF can be predicted using the converged values by making use of the number of user writes, obtained with V_H and V_{G_1} in S_n as user writes are sent to either *HOT* and G_1, and GC writes, obtained by summing the number of blocks in V_{G_2} and V_{G_3} at S_n. Thus, for our Fig. 7 example, WAF at S_n is estimated to be 1.63 (= (27.6+11.8+4.7+16.5)/(27.6+11.8)).

To validate WAFs predicted from MCAM, we compare measured and predicted WAF values for 50 randomly created group configurations. For evaluation, we make use of the YCSB-A and Varmail benchmarks. We first run the benchmark in a prototype of MiDAS implemented in a real-world system (see §5.2 for more details) and measure WAF values for the 50 configurations. While executing the benchmark, we also collect I/O traces and measure the average transition probabilities between segment groups. Then, we replay the collected traces on an MCAM simulator configured with the transition probabilities that we measured. We find that MCAM only shows an average error rate of 0.84% and 0.7%, with a maximum error rate of 2.82% and 2.33% for YCSB-A and Varmail, respectively, which confirms that, given a group organization, MCAM produces accurate WAF predictions.

To predict WAF from MCAM, however, we must provide the transition probabilities between segment groups. In the next subsection, we show how MiDAS estimates transition probabilities and how they are supplied to MCAM at runtime.

4.4 Estimating Transition Probabilities

To estimate the transition probabilities supplied to MCAM, we introduce UID (Update Interval Distribution). Fig. 8 illustrates the UID model, where the x-axis is the update interval and the y-axis represents the probability that blocks have the corresponding update interval. That is, UID is a probability mass function (PMF) of the update intervals of user-written blocks. How UID is obtained is described in §5.1, but in the meantime, we assume we have the UID, such as Fig. 8.

UID is used to obtain the probability of whether a block remains valid after a specific period of time, which can be directly translated into a transition probability for MCAM.

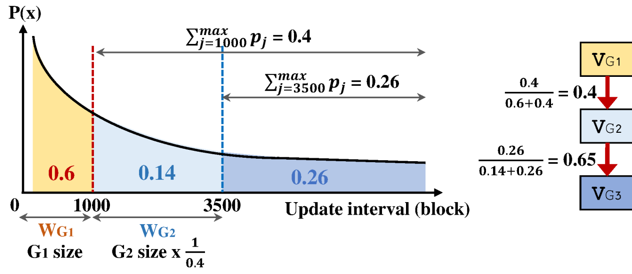


Fig. 8. Estimating transition probabilities using UID

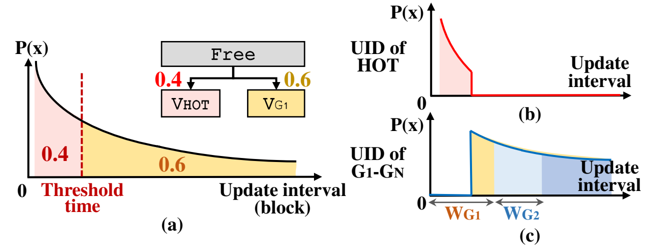


Fig. 9. Estimating transition probabilities, including *HOT*

We describe how this is done using the example in Fig. 8. Recall the unit of time in our system is defined as the number of user-written blocks. For simplicity sake, let us for now ignore *HOT* and assume only three segment groups, G_1, G_2, and G_3 each of size 1,000 exists, and that each segment is of 100 blocks. (We shall come back to *HOT* later.) User-written blocks are first sent to G_1 and then only valid blocks are moved to G_2. After 1,000 segments are written to G_1, G_1 becomes full and the victim segment at the tail of the group is selected for GC. (Recall from Fig. 6 that MiDAS uses the FIFO victim selection policy.) The number of live blocks moved to G_2 can be calculated based on UID. As the UID represents the probability of a user-written block being invalidated after a particular time interval, the sum of the probabilities of the update interval ranging from 1 to 1,000 is the probability of the blocks being invalid after 1000 writes. Let us assume, from Fig. 8, that this is 0.6, which means the transition probability from G_1 to G_2 is 0.4. Thus, out of the 100 blocks in the victim segment, 40 is transitioned to G_2.

In a similar manner, we can obtain the transition probability between G_2 and G_3. However, to fill up G_2, an additional 2,500 user-writes, that is, time steps, need to happen as only 40% of the user-written blocks are eventually sent to G_2. Once G_2 is filled with blocks, the segment at the tail of G_2 is selected as the victim. Now, let us consider how many valid blocks exist in this victim segment. To explain this, we define a new term, *waiting period* (denoted W_{G_i}), which refers to the number of user-written blocks required to fill up a specific segment group G_i. For our example, $W_{G_1} = 1000$, while $W_{G_2} = 2500$. Essentially, W_{G_i} is the elapsed time (*i.e.*, the number of user-written blocks) from when a new block comes into group G_i to when the block is evicted from the group. Now, given the UID for our workload, the sum of the probability measures for the period of W_{G_i} is the probability of the block becoming invalid. For our example, let us assume W_{G_2} is 0.14 (Fig. 8). Then, the sum of probability measures that the block remains valid is 0.26 ($= 0.4 - 0.14$). Thus, the total expected number of valid blocks expected to be moved to G_3 from the victim segment is 65 ($= 100 \times 0.65$) as the transition probability is 0.65 ($= \frac{0.26}{0.14+0.26}$).

Generalizing in this manner, the transition probability from $\mathtt{V}_{\mathtt{G_i}}$ to $\mathtt{V}_{\mathtt{G_{i+1}}}$, $T_{V_{G_i} \to V_{G_{i+1}}}$, is given by Eq. (1) (but not for $i = N$), where p_j is the probability for update interval j, max is the maximum update interval of UID, and W is the sum of the waiting periods of the groups, G_1 through G_{i-1}.

$$T_{V_{G_i} \to V_{G_{i+1}}} = \frac{\sum_{j=W+W_{G_i}}^{max} p_j}{\sum_{j=W}^{max} p_j}, \tag{1}$$

$$\text{where } W = \begin{cases} \sum_{k=1}^{i-1} W_{G_K} & \text{if } i > 1 \\ 0 & \text{otherwise.} \end{cases}$$

Transition probabilities including *HOT*: Let us now bring back $\mathtt{V_H}$ into the picture. To do that, we need to consider the transition probabilities from `Free` to $\mathtt{V_H}$ and from `Free` to $\mathtt{V}_{\mathtt{G_1}}$. Recall from Fig. 7 that the sum of the two must be 1.0. As mentioned earlier, user-written blocks are sent to either *HOT* or $G1$, and this decision is made by referring to the threshold time. Based on this, we obtain the transition probability from `Free` to $\mathtt{V_H}$, $T_{Free \to V_H}$, by summing the probabilities of the update intervals shorter than the threshold time in UID (0.4 in Fig. 9(a)). Thus, the transition probability from `Free` to $\mathtt{V}_{\mathtt{G_1}}$, $T_{Free \to V_{G_1}}$, is $1 - T_{Free \to V_H}$. Now, a keen reader will remember that blocks are sent to *HOT* only when it is observed that the latest update interval is less than the threshold time three times. Thus, the above explanation is not entirely correct. However, we find that not taking this into account still keeps the prediction accuracy within a maximum of 5% error. Thus, we make use of the above approximation.

Now returning back to the process that calculates the $T_{V_{G_i} \to V_{G_{i+1}}}$, we now exclude the probabilities for update intervals that are shorter than the threshold time by dividing the UID into two, UID for *HOT* and UID for G_1–G_N, as depicted in Figs. 9(b) and (c). As mentioned above, user-written blocks with update intervals shorter than the threshold time are sent to *HOT*. All blocks are also assumed to be invalidated in *HOT*. Thus, for UID of G_1–G_N, the probabilities for update intervals that are shorter than the threshold time are 0. We also need to consider the transition probability of `Free` to $\mathtt{V}_{\mathtt{G_1}}$ when calculating W_{G_1}, resulting in the equation W_{G_1} = size of $G_1 \times \frac{1}{T_{Free \to V_{G_1}}}$. Taking the same example where G_1 size is 1,000 blocks and $T_{Free \to V_{G_1}}$ is 0.6, only 60% of the user-written blocks are sent to G_1. Thus, W_{G_1} increases to 1,667, not 1,000, as $W_{G_1} = 1{,}000 \times \frac{1}{0.6}$. For the rest of the groups, we go through the same process as before to get their W_{G_i}. Finally, the transition probabilities for $T_{V_{G_i} \to V_{G_{i+1}}}$ can be obtained using Eq. (1).

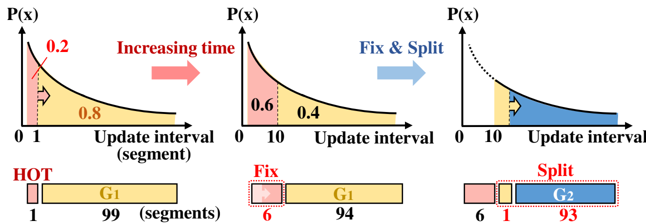


Fig. 10. Finding best group configuration with GCS

Transition probability for G_N: As the last group G_N comprises blocks of various ages as well as transitions to itself, the above analysis does not hold for G_N. Thus, to predict W_{G_N}, we make use of the analytical model proposed by Desnoyers [15][3]. This model accurately predicts WAF for techniques without data separation (*e.g.,* PageFTL [18]), where user-written blocks and GC copied blocks are placed in the same group, which is what is happening to the last group in MiDAS.

4.5 Configuring Groups with MCAM and UID

We now explain the group configuration selection (GCS) algorithm that finds the most appropriate group configuration. Given a specific group configuration, UID is able to compute transition probabilities between groups. By feeding the probabilities to MCAM, the expected WAF of the given configuration can be estimated as discussed in §4.3. Using UID and MCAM, GCS explores various group configurations to find the most appropriate one. Exploring every possible group configuration, however, is infeasible because of the huge exploration space and high computation cost.

Deciding the number of groups and their sizes: GCS takes a greedy heuristic approach to find a sufficiently good solution in reasonable time. GCS has two phases: (*i*) it roughly decides the number of groups and group sizes and then, (*ii*) fine-tunes the size of each group.

In the first phase, GCS begins with two segment groups: HOT and G_1. The primary objective of group partitioning is to ensure that for the blocks assigned to HOT, as many as possible are invalidated before eviction. The size of the HOT group needs to be carefully decided to provide sufficient time for written blocks to be invalidated before eviction. To achieve this, GCS first assigns data blocks with update intervals shorter than one segment time (the minimum unit of UID) to HOT and the rest are assigned to G_1, as depicted in Fig. 10. This figure shows an example where the sum of the probabilities within the one segment interval is 0.2. Since the unit of space allocation is a segment, GCS assigns one segment to HOT even if the suggested group size is only a few blocks smaller than a single segment.

Now that GCS has the initial sizes of HOT and G_1, it computes WAF using MCAM as explained in §4.3. GCS repeats the above step while increasing the time by one segment until a reduction in WAF is no longer observed. This is depicted in the leftmost to middle figure transition in Fig. 10. At this point, the group sizes for HOT and G_1 are determined. With HOT size fixed, the same steps are recursively repeated on G_1, that is, it splits G_1 into two groups, G_1 and G_2, and the size of G_1 increases until the observed WAF is minimized. This is depicted in the middle to rightmost figures in Fig. 10. GCS continues to split groups until no noticeable WAF reduction $> 0.5\%$ is observed over five consecutive splits.

After the number of groups and group sizes are decided, GCS goes through the second phase where the group sizes are fine-tuned. By allocating segments from the earlier groups, that is, from HOT to G_i in increasing i order, the first phase, prioritizes the earlier groups. This results in high WAF of the last group G_N as it is unlikely to get sufficient segments. To mitigate this, MiDAS reassigns segments by transferring one segment from HOT to G_N. Then, the WAF is computed. If the newly computed WAF is smaller than the old one, the movement is confirmed. Otherwise, the movement is reverted back. This is done for the remaining groups, G_i to G_N starting from $i = 1$ up to $N-1$, until no more WAF reduction is observed.

While GCS requires moderate computation, it is invoked only when a new UID is built to adapt to changes of the workloads. Moreover, it does not affect foreground jobs (*e.g.,* write or read requests) as it is performed in the background.

Updating group configuration: The accuracy of predicting transition probabilities using UID will drop if the workload pattern changes over time. Deciding the size of a group with an incorrect UID may exacerbate overall WAF. To address this, MiDAS periodically generates a new UID and uses the new one if it provides higher accuracy. More specifically, MiDAS divides time into epochs of the same length and maintains two UIDs, one that is currently being used, which was generated in the previous epoch, and one that we generate in the current epoch. At the end of each epoch, MiDAS estimates the lowest possible WAF for the new group organization using the newly created UID and compares it to the WAF of the existing group organization with the old UID. If the WAF is not reduced by more than a certain threshold MiDAS continues to use the UID. However, if the difference exceeds the threshold, MiDAS adopts the new group organization based on the new UID. Currently, the threshold is set to 5%. Note that for the first epoch where no information about the workload is available, MiDAS simply employs the age-based MiDA policy with the CB victim selection policy.

The length of the epoch may have an impact on UID accuracy. A short epoch enables quick adaptation to changes in workload patterns but can lead to an accuracy drop due to the small amount of information. A lengthy epoch, on the other hand, may increase the accuracy of UID, but will make MiDAS less sensitive to workload changes. We experimentally determine the epoch to be four times the storage capacity. We evaluate the impact of the epoch length in §6.3.

Handling irregular I/O patterns: While MiDAS provides high accuracy in predicting transition probabilities using UID

[3] The specific model is presented in our supplemental material [41].

when workloads have regular I/O patterns, in reality, not all I/O patterns are regular. In many of the traces that we analyzed, we observe workloads with high I/O fluctuation and sudden unexpected I/O pattern changes over time. In such cases, MiDAS may fail to predict the future behavior of the workload using UID, resulting in high WAF. Thus, MiDAS takes a different approach, which we elaborate below.

First, MiDAS needs to check for irregularities. This is done by regularly checking how much the predicted and actual transition probabilities differ. This is easy to do as we have the predicted transition probabilities derived from UID and the actual transition probabilities can be obtained by keeping track of the number of valid blocks moved from one group to another. Once a high error rate is detected between groups, say between G_k and G_{k+1}, MiDAS gives up on adjusting group sizes for all groups beyond G_k and simply merges the groups from G_k to G_N into G_k. This is because the low accuracy between G_k and G_{k+1} will have a cascading impact on the accuracy of the subsequent groups. Then, MiDAS falls back to the simple age-based MiDA technique as we did when no workload information was available. This is maintained until an up-to-date UID is generated at the next epoch and can be used to find a new group configuration. Currently, this fallback mechanism is invoked when the error rate between the predicted and actual transition probabilities exceeds 10%.

5 Implementation and Experimental Setup

In this section, we discuss some implementation details including methods used to optimize memory. We then detail our experimental setup.

5.1 Implementation and Resource Overhead

In MiDAS, a chain of the cold groups, each organized as a simple FIFO queue, is managed using a linked list that consumes little memory. When the group configuration is changed, the segments containing data blocks do not physically move across the groups. Instead, only pointers that point to the physical location of each segment in the queue are moved. This allows for simple adjustment of group configurations without any data copy overhead. For example, when merging two groups, the pointers are relocated to one of the queues. Conversely, when dividing a single group into two, MiDAS initially creates a new queue and a free segment for the newly created group. Subsequently, the pointers from the segments in the original group are moved upon the activation of GC in that group.

MCAM requires moderate CPU cycles (we will discuss this in §6.4) but only needs to keep a few parameters (*i.e.,* transition probabilities and block counts) that require minimal memory. For *HOT* and UID, however, MiDAS has to keep various data structures. We discuss optimizations that we conduct to reduce memory overhead.

Hot filter: Recall that MiDAS promotes a data block to *HOT* when its update interval is found to be within the threshold time three times. To manage this, MiDAS maintains a 2-bit hot filter per block in memory. We find that this requires less than 60MiB per 1TiB storage.

Table 3: Characteristics of each workload

Workloads	FIO-H & -M	Varmail	YCSB-A & -F	TPC-C	Alibaba	Exchange
Notation	F-H & F-M	V	Y-A & Y-F	T-C	Ali	Ex
Write size (TiB)	15.1	14.9	16.4 & 18.0	16.1	Up to 2.8	Up to 2.9
Device size (GiB)	128				40-200	40
Request distribution	Zipfian (1.0 & 0.8)	Zipfian			-	-

Constructing UID: To construct UID, MiDAS uses two key data structures: a timestamp table and an interval count table. The timestamp table records timestamps of block updates to compute the update intervals of data blocks. To save memory space, instead of keeping track of all data blocks, we sample only a subset for timestamp monitoring, with a sampling rate of 0.01 (one in every 100 blocks). This reduces the timestamp table size to 10.3MiB per 1TiB storage. The interval count table keeps track of the number of blocks for specific update intervals. To reduce the size of the interval count table, we use a coarse-grained update interval unit of 16K blocks rather than a block unit. Blocks with update intervals falling within the range of [1, 16K blocks] are considered to have the same update interval. In this way, the interval count table reduces to 256KiB in size per 1TiB storage. The accuracy degradation by this optimization is less than 3%.

5.2 Experimental Setup

We carry out experiments using both trace-driven simulations and a real SSD prototype. An existing SSD simulator [12], with block I/O traces collected from various environments, is used to quickly evaluate key performance metrics (*e.g.,* WAF) of the GC techniques. We also use a real SSD prototype, where MiDAS is implemented within the FTL, to measure I/O performance and the overheads associated with the CPU and memory for system execution. Our SSD prototype is equipped with a quad-core ARM Cortex-A53 and 256MiB DRAM for indexing memory. The SSD platform employs a 256GiB custom flash array card with 16KiB flash page size and 128 pages per block and 8×8 channel. Unless otherwise stated, the over-provisioning ratio is set to 7.3% as this is typical in commercial systems [26, 39]. For both the simulator and SSD prototype, we implement the CAT, AutoStream, MiDA, and SepBIT SOTA GC techniques.

The following write-intensive workloads are used for our evaluations: FIO workloads [4] whose data access pattern follows Zipf distribution with theta values of 1.0 (denoted FIO-H) and 0.8 (denoted FIO-M), where the former and latter, respectively, represent workloads with skewed and relatively uniform data access patterns; Varmail, the most write-intensive workload in the Filebench benchmark [43]; YCSB-A and -F, the write-intensive workloads of the YCSB benchmark [13]; TPC-C [14] running on MySQL [46]; 9 Exchange traces from Microsoft Enterprise Traces [35]; and 25 write-

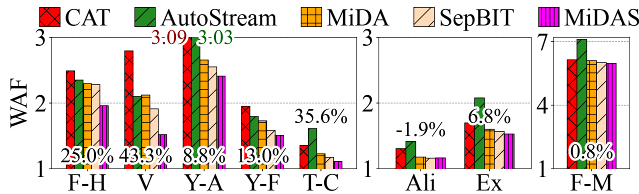


Fig. 11. Overall WAF of each technique (the numbers represent the improvement on WAF by MiDAS relative to SepBIT)

intensive traces from the Alibaba Cloud I/O traces [30], whose LBA ranges are smaller than 256GB. Note that the traces of Exchange and Alibaba, individually, are relatively small. For these two, each trace is run individually and the average results are reported. The workload summary and details are shown in Table 3, and we use the abbreviated notations there to denote the workloads in reporting the results. Before running the workloads, we fill up 92.7% of the SSD space with data to make the SSD quickly reach a steady-state condition that invokes GC regularly. GC is triggered when free space drops below 0.1% of the total SSD capacity.

6 Experimental Results

6.1 Comparison of GC Efficiency

We first evaluate GC efficiency by comparing the WAFs of the various techniques. All experiments to measure WAFs are conducted using the trace-driven simulator. All techniques, except for MiDAS, adopt CB, the most efficient victim selection policy. Fig. 11 shows WAF for each technique. The numbers on or above each workload bar represent the improvement on WAF (%) by MiDAS relative to SepBIT, which is the best performing SOTA technique.

With FIO-H, where data access is highly skewed, a noticeable WAF reduction is observed. This is because many hot blocks are filtered out by *HOT*, which reduces the number of valid block copies during GC. By setting the hot boundary precisely and adjusting the number of cold groups and their sizes, MiDAS outperforms the SOTA techniques. (A breakdown of each contributing factor is given and discussed in more detail in §6.2.) On the other hand, with FIO-M, where almost all of the data blocks have similar update intervals with little variations, WAF reduction by hot-cold separation and group size adjustment is limited.

For Varmail, YCSB-A and -F, and TPC-C, MiDAS reduces WAF by 8.8–43.3% compared to SepBIT. These workloads not only have numerous hot blocks but also a significant number of warm blocks. MiDAS captures these characteristics with UID and then, changes the group configuration accordingly. As a result, MiDAS ensures that the warm blocks are invalidated before eviction, significantly reducing unnecessary data block copies (see §6.2).

In case of the Alibaba workloads, MiDAS and SepBIT show similar WAF, with MiDAS showing worse WAF on average. Recall that the Alibaba workloads are relatively short, showing write sizes of 2.23 epochs on average, so most of the traces end before the obtained UID can reap its benefits. Moreover, Alibaba workloads show irregular I/O patterns that increase the inaccuracy of predictions. This leads MiDAS to incorrect group configurations or to simply fall back to the MiDA technique. Thus, we see that WAF of MiDAS and MiDA are also very similar. Exchange also is composed of 9 small workloads and we observe similar irregular I/O patterns. Still, MiDAS is able to reduce WAF by 6.8% compared to SepBIT.

Overall, for the workloads we considered, MiDAS reduced WAF by an average of 25%. If we exclude the workloads with irregular patterns, Alibaba and Exchange, and the one with relatively uniform data access, FIO-M, the reduction on average is 34.7%.

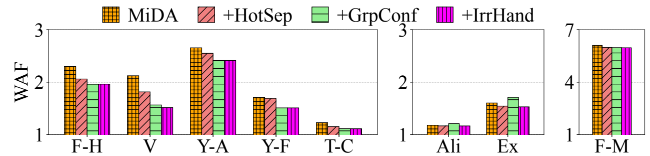


Fig. 12. Impact of individual components of MiDAS

6.2 Impact of Each Component of MiDAS

We analyze the impact of the individual design components of MiDAS, that is, (*i*) hot block separation (§4.2), (*ii*) group configuration (§4.3–§4.4), and (*iii*) irregular I/O pattern handling (§4.5). To observe the effect of adding each component, we start off with MiDA, the basic design that MiDAS takes from. Then, we add each design component denoted, +HotSep, +GrpConf, +IrrHand, respectively, one after the other, observing the performance as each component is added.

Fig. 12 shows how each component improves (or sometimes worsens) WAF. We distinguish the workloads into two groups, Group 1, those more typical workloads with some skewness and regularity in data access and the other, Group 2, those whose I/O patterns are largely irregular (Exchange, Alibaba) and that with low skewness in access (FIO-M), as they show distinctly different influence.

Those in Group 1 show +HotSep and +GrpConf, individually, bringing about considerable gains. As explained in §6.1, +HotSep improves the performance of FIO-H by separating hot blocks. +GrpConf is effective in Varmail, YCSB-A and -F, and TPC-C by organizing groups to give sufficient space to warm blocks. In contrast, +IrrHand shows minimal, if any, benefits. We see slight gains with Varmail and TPC-C, where small irregular patterns are detected, which is reflected in Table 4 as will be explained below.

The results for Group 2 are quite different. We see that the effect of +HotSep is smaller than for Group 1. In case of FIO-M, the impact of each component is limited as the block access is less skewed and has no irregular patterns. Thus, recall from Fig. 11 that, for FIO-M, all the techniques, except Autostream, showed similar WAF. For Alibaba and Exchange

Table 4: Accuracy of UID in predicting transition probabilities

Workloads	F-H	V	Y-A	Y-F	T-C	Ali	Ex	F-M
Avg. error (%)	1.82	6.90	2.33	1.78	4.81	**11.44**	**8.16**	1.33
Avg. error (%) w/ +IrrHand	1.82	1.97	2.33	1.78	2.38	4.67	3.36	1.33

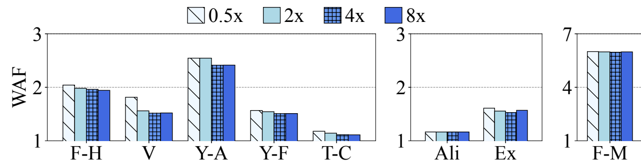
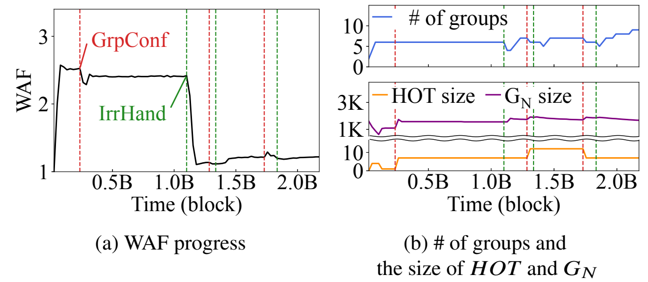


Fig. 13. Impact of length of epoch on generating UID



(a) WAF progress

(b) # of groups and the size of HOT and G_N

Fig. 14. MiDAS adapting to workload change

+GrpConf has a profound negative effect due to their irregular I/O patterns that make finding suitable group configurations difficult. The first row in Table 4 lists the average errors between UID predicted and actual transition probabilities. The second row is the same error but when +IrrHand is applied. We see these values coming down for Varmail, TPC-C, Alibaba, and Exchange. Before +IrrHand is applied, MiDAS is making inaccurate decisions in configuring groups based on these erroneous predictions. Hence, when +GrpConf is applied, WAF increases for Alibaba and Exhange. For Varmail and TCP-C, the inaccuracy is not as serious, so we see improvements after applying +GrpConf. We also see that +IrrHand mitigates the negative effect of irregular I/O patterns.

6.3 Miscellaneous Results

Impact of epoch length: We consider epoch lengths of 0.5×, 2×, 4×, and 8× of the storage capacity (128GiB) and observe the WAFs. Recall that all the earlier experiments were performed with an epoch length of 4×128GiB. As shown in Fig. 13, workloads are largely insensitive to epoch lengths of 4× and above with the exception of Exchange whose irregular patterns have strong influence. In contrast, shortening the length negatively affects most workloads.

Adapting capability: We evaluate how well MiDAS responds to changes in workload patterns. To see this, we run YCSB-A from 0 to 1.1 billion (B) time and then switch to TPC-C until the end of the experiment. While running, we measure WAF, the number of groups, and HOT and G_N sizes. Fig. 14(a) depicts how WAF changes over time. Adapting group configuration (+GrpConf) occurs six times during the entire process, while irregular pattern handling (+IrrHand) is invoked five times in each epoch. The points where MiDAS adopts GrpConf and IrrHand are highlighted by the dotted lines. The group configuration set at 0.25B remains unchanged until 1.1B due to stable transition probabilities in the YCSB-A workload (see Table 4). Once TPC-C begins at 1.1B, the I/O pattern changes dramatically, with IrrHand merging groups with high error rates to optimize WAF.

Fig. 14(b) shows how the number of groups, the HOT and G_N sizes evolve. We observe that when IrrHand is invoked, the size of G_N changes, while HOT remains unchanged due to low error rates. We conclude that MiDAS is effectively accommodating the needed changes to the groups (number and size) according to the changes in the workload.

Comparison with ORA: We pointed out the limitations of the existing techniques in data placement and group size decisions by comparing them with ORA in Fig. 4 and Table 2, respectively. We now compare MiDAS with ORA to show how efficiently MiDAS addresses these limitations. Fig. 15(a) shows the distribution of data blocks assigned to groups for YCSB-A and the group size of each group (numbers on each bar). The figure also displays the ratios of block categories, C_1, ... C_6, decided by ORA. Note that the number of groups in MiDAS is the same as that of ORA and that this was reached through adjustments. We also observe that most of the hot blocks categorized as C_1 are assigned to HOT. MiDAS also provides sufficient space to HOT so that hot blocks can be invalidated before being evicted to G_1. However, we observe that, though much more accurate than the other techniques, data blocks in cold categories (*i.e.,* C_4–C_6) account for non-trivial portions in G_1–G_2. This is owing to the age-based migration policy of MiDAS that writes data blocks to the former groups and then moves live blocks to the subsequent groups. Despite such errors, age-based migration enables us to efficiently segregate cold blocks in the latter groups (G_3–G_6), helping reduce overall WAF.

6.4 Experiments on SSD Prototype

To evaluate throughput and CPU utilization that cannot be measured from the trace-driven simulator, we implement a proof-of-concept prototype of MiDAS in a real-world SSD and carry out a set of experiments. We implement PageFTL, MiDA, and SepBIT in the SSD platform. Six workloads, Varmail, YCSB-A, YCSB-F, TPC-C, Alibaba and Exchange, are used for the throughput experiments. For Alibaba and Exchange, we report the average throughput of the subtraces.

As shown in Fig. 15(b), MiDAS achieves 2.55×, 1.24×, 1.15× higher throughput, respectively, than PageFTL, MiDA, and SepBIT, on average. Notably, for Varmail, MiDAS, respectively, achieves 3.2×, 1.4× and 1.3× gains. This is because MiDAS can significantly reduce writes by GC compared to other techniques, evidenced by the WAF values of 4.26, 2.12, 1.91, and 1.52 for PageFTL, MiDA, SepBIT, and MiDAS, respectively. These are in line with the simulation results. Moreover, for Alibaba workloads, MiDAS improves

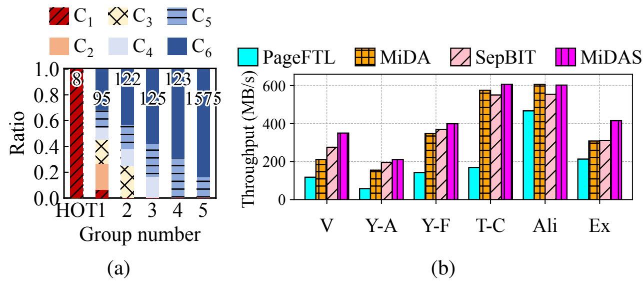


Fig. 15. (a) MiDAS block distribution, (b) Throughput result

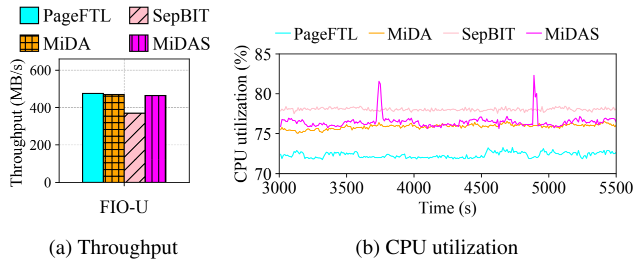


Fig. 16. Results from our SSD prototype for FIO-U

the throughput by 8.7% compared to SepBIT despite MiDAS showing higher WAF than SepBIT.

We also conduct experiments to measure the CPU overhead. To remove the GC impact, we use the FIO workload with uniform distribution (FIO-U) configured so that the WAFs of all the techniques become almost identical (1.2) and allow only 60% of the entire device to be composed of valid blocks. We use this configuration to assess I/O throughput when the techniques handle ordinary user requests without being impacted by GC. Fig. 16(a) shows that all the techniques, except for SepBIT, exhibit similar throughput as they have the same WAF. SepBIT shows the lowest throughput among the techniques. This is because it incurs high CPU utilization to maintain the queue as well as to lookup the queue for every user-written block in order to detect hot blocks. Note that the negative impact on CPU utilization is particularly significant in FIO-U where user-written blocks are dominant. For garbage-collected blocks, SepBIT does not need to look up the queue, so CPU overhead is much lower when the GC is frequently triggered.

Fig. 16(b) shows the CPU utilization of FIO-U over time. MiDAS shows, on average, only 4.2% and 0.6% higher CPU utilization compared to PageFTL and MiDA, respectively. MiDAS is designed to require only a few CPU cycles in common paths such as reads, writes, and GC. However, we notice sharp increases in CPU utilization at around 3740s and 4890s. This is where MiDAS starts to run MCAM to find the best group configuration. However, its impact on performance is minimal as this is run in the background. As shown in Fig. 16(a), MiDAS provides similar throughput as PageFTL and MiDA.

7 Discussion

In the current stage, our work is limited to a case study for flash-based SSDs. However, it can be adapted to other log-structured systems. For instance, for ZNS SSDs, MiDAS can be integrated into a zoned storage backend (*e.g.*, ZenFS [6]) to reduce the host's GC overhead, similar to SepBIT [44]. The segment size in MiDAS is not fixed and thus can be adjusted to match the characteristics of ZNS SSDs.

However, there could be several hurdles when adapting to other log-structured systems, such as the LSM-tree. LSM-trees have chain-like structures that merge-sort data through progressively larger layers [36, 40]. MiDAS's structure is similar, where live blocks migrate to subsequent groups via GC, and its group configuration could be integrated into LSM-tree level design. However, a new challenge arises because LSM-trees keep objects in a sorted order, conflicting with MiDAS's assumption that blocks in a segment share similar age. Additionally, the significantly larger key range of LSM-tree objects compared to the LBA range leads to an expanded timestamp table, thereby causing an increased memory footprint. We plan to address these problems as part of our future work.

Many researchers have discussed write amplification reduction techniques in log-structured systems like ZNS and LSM-tree [5, 19, 32, 52]. However, research that dynamically applies group configuration based on the workload pattern in log-structured systems other than flash-based SSDs has not been explored in great depth. MiDAS can provide useful insights on how to effectively apply group configurations within general log-structured systems.

8 Conclusion

In this paper, we presented MiDAS, a systematic solution to mitigate GC overhead for log-structured systems. MiDAS employs a chain-like structure to organize data by age and minimize data movement between groups using analytical models, UID and MCAM. It also isolates hot blocks within a designated hot group and dynamically adjusts its size against cold groups in a manner that minimizes overall GC costs. Our experiments demonstrated that MiDAS outperforms existing techniques, achieving 25% reduction in WAF and 54% higher throughput, on average for the workloads considered, all while being memory-efficient and consuming fewer CPU cycles.

Acknowledgments

We thank our shepherd, professor Ming-Chang Yang, and the anonymous reviewers for all their helpful comments. This work was supported by SNU-SK Hynix Solution Research Center (S3RC), the National Research Foundation of Korea (NRF-2018R1A5A1060031), the MOTIE (Ministry of Trade, Industry & Energy) (1415181081), KSRC (Korea Semiconductor Research Consortium) (20019402), and NSF grant (2312785). (Corresponding author: Sungjin Lee)

References

[1] Abutalib Aghayev and Peter Desnoyers. Skylight—A window on shingled disk operation. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 135–149, 2015.

[2] Abutalib Aghayev, Theodore Ts'o, Garth Gibson, and Peter Desnoyers. Evolving Ext4 for Shingled Disks. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 105–120, 2017.

[3] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark S. Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *Proceedings of the USENIX Annual Technical Conference*, pages 57–70, 2008.

[4] Jens Axboe. FIO: Flexible I/O Tester Synthetic Benchmark. https://github.com/axboe/fio, 2023.

[5] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *Proceedings of the USENIX Annual Technical Conference*, pages 363–375, 2017.

[6] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *Proceedings of the USENIX Annual Technical Conference*, pages 689–703, 2021.

[7] Pierre Brémaud. *Markov chains: Gibbs fields, Monte Carlo simulation, and queues*, volume 31. Springer Science & Business Media, 2001.

[8] Werner Bux and Ilias Iliadis. Performance of Greedy Garbage Collection in Flash-Based Solid-State Drives. *Performance Evaluation*, 67(11):1172–1186, 2010.

[9] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, pages 181–192, 2009.

[10] Mei-Ling Chiang and Ruei-Chuan Chang. Cleaning Policies in Mobile Computers Using Flash Memory. *Journal of Systems and Software*, 48(3):213–231, 1999.

[11] Mei-Ling Chiang, Paul C.H. Lee, and Ruei-Chuan Chang. Using Data Clustering to Improve Cleaning Performance for Flash Memory. *Software: Practice and Experience*, 29(3):267–290, 1999.

[12] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. Lightstore: Software-defined Network-attached Key-value Drives. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 939–953, 2019.

[13] 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*, pages 143–154, 2010.

[14] The Transaction Processing Council. TPC-C Benchmark. https://www.tpc.org/tpcc, 2021.

[15] Peter Desnoyers. Analytic Models of SSD Write Performance. *ACM Transactions on Storage*, 10(2):1–10, 2014.

[16] Samsung Electronics. Multi-Stream Write SSD. *Flash Memory Summit*, 2016.

[17] Garth Gibson and Greg Ganger. Principles of Operation for Shingled Disk Devices. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems*, 2011.

[18] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 229–240, 2009.

[19] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 147–162, 2021.

[20] John A. Hartigan and Manchek A. Wong. Algorithm AS 136: A K-means Clustering Algorithm. *Applied Statistics*, 28(1):100–108, 1979.

[21] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The Multi-streamed Solid-State Drive. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems*, 2014.

[22] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: Transactional FTL for SQLite Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 97–108, 2013.

[23] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A Flash-Memory Based File System. In *Proceedings of the USENIX Annual Technical Conference*, pages 155–164, 1995.

[24] Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, and Yookun Cho. A Space-efficient Flash Translation Layer for CompactFlash Systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[25] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. Fully Automatic Stream Management for Multi-Streamed SSDs Using Program Contexts. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 295–308, 2019.

[26] Kingston. Understanding SSD Over-provisioning (OP). https://www.kingston.com/en/blog/pc-performance/overprovisioning, 2019.

[27] Kevin Kremer and André Brinkmann. FADaC: A Self-Adapting Data Classifier for Flash Memory. In *Proceedings of the ACM International Conference on Systems and Storage*, pages 167–178, 2019.

[28] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 273–286, 2015.

[29] Jan Van Leeuwen. *Handbook of theoretical computer science (vol. A) algorithms and complexity*. Mit Press, 1991.

[30] Jinhong Li, Qiuping Wang, Patrick P.C. Lee, and Chao Shi. An In-Depth Analysis of Cloud Block Storage Workloads in Large-Scale Production. In *Proceedings of IEEE International Symposium on Workload Characterization*, pages 37–47, 2020.

[31] Seung-Ho Lim, Hyun Jin Choi, and Kyu Ho Park. Journal Remap-based FTL for Journaling File System with Flash Memory. In *Proceedings of the International Conference on High Performance Computing and Communications*, pages 192–203, 2007.

[32] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 133–148, 2016.

[33] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 1–16, 2012.

[34] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: An Elastic Zoned Namespace for Commodity ZNS SSDs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 461–477, 2023.

[35] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In *Proceedings of the ACM European Conference on Computer Systems*, page 145–158, 2009.

[36] 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.

[37] Hyunseung Park, Eunjae Lee, Jaeho Kim, and Sam H. Noh. Lightweight Data Lifetime Classification Using Migration Counts to Improve Performance and Lifetime of Flash-Based SSDs. In *Proceedings of the ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 25–33, 2021.

[38] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[39] Samsung Electronics. Over-provisioning Benefits for Samsung Data Center SSDs. https://download.semiconductor.samsung.com/resources/white-paper/S190311-SAMSUNG-Memory-Over-Provisioning-White-paper.pdf, 2019.

[40] Russell Sears and Raghu Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 217–228, 2012.

[41] Seonggyun Oh, Jeeyun Kim, Soyoung Han, Jaeho Kim, Sungjin Lee, and Sam H. Noh. Supplemental Material of MiDAS. https://github.com/dgist-datalab/MiDAS/blob/main/MiDAS_supplemental_material.pdf, 2024.

[42] Radu Stoica and Anastasia Ailamaki. Improving Flash Write Performance by Using Update Frequency. *VLDB Endowment*, 6(9):733–744, 2013.

[43] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. *The USENIX Magazine*, 41(1):6–12, 2016.

[44] Qiuping Wang, Jinhong Li, Patrick P.C. Lee, Tao Ouyang, Chao Shi, and Lilong Huang. Separating Data

via Block Invalidation Time Inference for Write Amplification Reduction in Log-Structured Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 429–444, 2022.

[45] Western Digital. Western Digital Ultrastar DC ZN540 Data Sheet. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/collateral/data-sheet/data-sheet-ultrastar-dc-zn540.pdf, 2021.

[46] Wikipedia. MySQL. https://en.wikipedia.org/wiki/MySQL, 2023.

[47] Michael Wu and Willy Zwaenepoel. ENVy: A Non-Volatile, Main Memory Storage System. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, page 86–97, 1994.

[48] Gala Yadgar, Moshe Gabel, Shehbaz Jaffer, and Bianca Schroeder. SSD-Based Workload Characteristics and Their Performance Implications. *ACM Transactions on Storage*, 17(8):1–26, 2021.

[49] Jing Yang and Shuyi Pei. Thermo-GC: Reducing write amplification by tagging migrated pages during garbage collection. In *Proceedings of the IEEE International Conference on Networking, Architecture and Storage*, pages 1–8, 2019.

[50] Jing Yang, Shuyi Pei, and Qing Yang. WARCIP: Write amplification reduction by clustering I/O pages. In *Proceedings of the ACM International Conference on Systems and Storage*, pages 155–166, 2019.

[51] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. AutoStream: Automatic stream management for multi-streamed SSDs. In *Proceedings of the ACM International Systems and Storage Conference*, pages 1–11, 2017.

[52] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *Proceedings of the USENIX Annual Technical Conference*, pages 17–31, 2020.

A Artifact Appendix

Abstract

MiDAS is a migration-based data placement technique with adaptive group number and size configuration for reducing garbage collection overhead in various log-structured systems. Notably, MiDAS is currently implemented within the FTL of the real SSD prototype. For artifact evaluation, we provide our source code and the trace file. Please refer to the README file at https://github.com/dgist-datalab/MiDAS.

Scope

The artifact includes all the necessary source code required to run MiDAS as well as the FIO-based workload trace file used in this study. You can quickly test MiDAS using this trace.

Contents

We provide two Git repositories related to MiDAS: the SSD prototype-based implementation and the trace-driven simulation. If you evaluate MiDAS using the SSD prototype-based implementation, you can measure not only the WAFs but also I/O performance and the overhead associated with the CPU and memory for system execution. Meanwhile, using trace-driven simulations, you are limited to WAF evaluations only. Here, we will explain the contents based on the SSD prototype-based implementation. There are four main c files to run MiDAS:

- midas.c: adaptably changes group configuration and regularly checks irregular patterns at runtime (see §4.5).
- model.c: constructs UID, predicts WAF using MCAM, and runs GCS algorithm to find the best group configuration based on the observed workload patterns (see §4.3, §4.4 and §4.5).
- gc.c: selects a victim of GC based on the victim selection policy and moves live blocks into subsequent groups (see §4.1).
- hot.c: constructs a hot filter to separate hot blocks from cold blocks by monitoring incoming data blocks (see §4.2).

Hosting

We provide the public Git URLs, and the commit hashes for each repository used during artifact evaluation. MiDAS is implemented in both a real SSD prototype and trace-driven simulation. For quick testing of MiDAS, particularly for evaluating WAF, using the trace-driven simulation is sufficient. Additionally, we provide a public Zenodo URL for downloading the trace file of the FIO benchmark used in our evaluation.

- Emulated SSD prototype
  https://github.com/dgist-datalab/MiDAS
  `14be7bf7b01fad8db023622e4598fb9e30d8024f`
- Trace-driven simulation
  https://github.com/sungkyun123/MiDAS-Simulation
  `a22626529ad625eb4ddbab298752b9116fe6d05a`
- FIO-based workload trace file
  https://zenodo.org/record/10409599

Requirements

Hardware requirements. We use the Xilinx Virtex® UltraScale™ FPGA VCU108 platform and customized NAND flash modules. The customized NAND flash modules used in this paper are not publicly or commercially available. Therefore, you may need your own NAND modules compatible with VCU108 and adequate modifications to the hardware backend to replicate this work. Acknowledging the challenge for most researchers to replicate our experimental setup, we offer an alternative emulation of the prototype via a memory-based approach (RAM drive). DRAM must be larger than the device size of trace files + an extra 10% of the device size for the data structures and OP region to test the trace files. For example, you need 140GiB size of DRAM to run the trace file with a 128GiB device size.

Software requirements. There are little special software requirements to run MiDAS and you only need to install some packages using `apt`. The README file in the repository describes detailed instructions for the installation.

What's the Story in EBS Glory: Evolutions and Lessons in Building Cloud Block Store

Weidong Zhang, Erci Xu*, Qiuping Wang, Xiaolu Zhang, Yuesheng Gu, Zhenwei Lu, Tao Ouyang, Guanqun Dai, Wenwen Peng, Zhe Xu, Shuo Zhang, Dong Wu, Yilei Peng, Tianyun Wang, Haoran Zhang, Jiasheng Wang, Wenyuan Yan, Yuanyuan Dong, Wenhui Yao, Zhongjie Wu, Lingjun Zhu, Chao Shi, Yinhu Wang, Rong Liu, Junping Wu, Jiaji Zhu, Jiesheng Wu

Alibaba Group

Abstract

In this paper, we qualitatively and quantitatively discuss the design choices, production experience, and lessons in building the Elastic Block Storage (EBS) at ALIBABA CLOUD over the past decade. To cope with hardware advancement and users' demands, we shift our focus from design simplicity in EBS1 to high performance and space efficiency in EBS2, and finally reducing network traffic amplification in EBS3.

In addition to the architectural evolutions, we also summarize development lessons and experiences as four topics, including: (i) achieving high elasticity in latency, throughput, IOPS and capacity; (ii) improving availability by minimizing the blast radius of individual, regional, and global failure events; (iii) identifying the motivations and key tradeoffs in various hardware offloading solutions; and (iv) identifying the pros/cons of alternative solutions and explaining why seemingly promising ideas would not work in practice.

1 Introduction

Elastic Block Storage (EBS) service is a cornerstone in today's cloud [16, 18, 19]. In EBS, the storage service is in the form of virtual block devices with high performance, availability, and elasticity. The most outstanding characteristic of EBS architecture is the compute-to-storage disaggregation where the virtual machines (compute end) and disks (storage end) are not physically co-located but interconnected via datacenter networks.

In this paper, we start by revisiting the evolutions behind the three generations of EBS at ALIBABA CLOUD [16]. EBS1 marks our initial step in adopting the compute-to-storage philosophy. In EBS1, there are two notable design choices: in-place update from virtual disks (VDs) to physical disks, and the exclusive management of virtual disks. First, EBS1 directly maps a VD inside the virtual machine (VM) as a series of 64 MiB Ext4 files in the backend storage server. Moreover, EBS1 employs a fleet of stateless BlockServers to manage VDs where each VD is exclusively handled by a BlockServer. While EBS1 had been successfully deployed on more than 300 HDD-backed clusters, its limitations also unfolded. The straightforward virtualization led to severe space amplification and performance bottlenecks.

*Corresponding author.

We then developed EBS2 with two significant changes: the log-structured design, and VD segmentation. First, we employed the Pangu [35] distributed file system as our storage backend, and redesigned the BlockServers to convert VDs' all writes to sequential appends. By switching to a log-structured layout, EBS2 still used three-way replication for incoming writes but could transparently perform data compression and erasure coding (EC) in the background during garbage collection (GC). Moreover, EBS2 split VDs into finer segments (32 GiB each), thus shifting the mapping between VDs and BlockServers from VD level to Segment level. With the above two changes, EBS2 was able to reduce the space efficiency from 3 (i.e., three-way replication) in EBS1 to 1.29 on average in the field. Moreover, supercharged with SSDs, an EBS2-backed VD can achieve up to 1 M IOPS and 4,000 MiB/s throughput with 100 μs-level latency on average. Unfortunately, EBS2 also faced a significant challenge. That is, the traffic amplification factor increased to 4.69, namely 3 (foreground replication write) plus 1 (background GC read) and 0.69 (background EC/compression write).

Hence, we built EBS3 to reduce traffic amplification using online (i.e., foreground) EC/compression via two techniques: Fusion Write Engine (FWE), and FPGA-based hardware compression. FWE aggregates write requests from different segments (if necessary) to meet the size requirement of EC and compression. Moreover, EBS3 offloads the compute-intensive compression to a customized FPGA for acceleration. As a result, EBS3 can reduce the storage amplification factor from 1.29 to 0.77 (after compression) and the traffic amplification factor from 4.69 to 1.59 while still maintaining performance similar to EBS2. Since release, EBS3 has been deployed on more than 100 clusters, serving over 500K VDs.

Figure 1 outlines the chronological progression of Alibaba EBS since 2012. We highlight the time of major releases (i.e., EBS1 to EBS3), the integration of key techniques (e.g., Luna, our user-space TCP stack [46]) and the adoption of advanced hardware (e.g., Persistent Memory in EBSX). The evolution of EBS demonstrates a shift in focus from performance to space

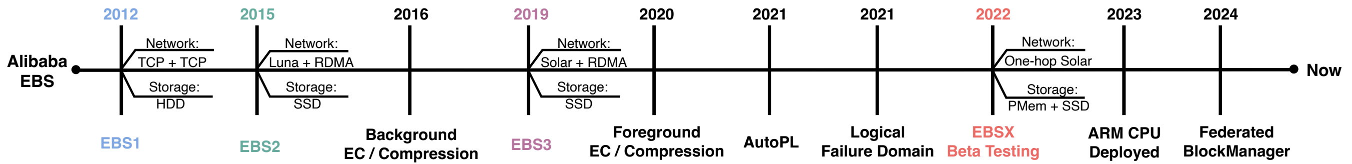


Figure 1: Alibaba EBS Timeline

and traffic efficiency. Nevertheless, simply altering the high-level architecture is not enough. Next, we further discuss our lessons in building a high-performance and robust EBS from four perspectives, including elasticity, availability, hardware offloading, and alternative solutions.

One representative feature of a cloud block store is elasticity—providing VDs with varying performance and capacity levels. There are two key aspects: identifying boundaries and achieving fine-grained tuning. First, we discover that the average and tail latency are dominated by different factors—hardware overhead and software processing, respectively. Thus, we build corresponding solutions including EBSX, a one-hop architecture backed by persistent memory for minimizing average latency, and the use of dedicated threads for I/O to alleviate tail latency. Furthermore, we realize that throughput and IOPS are bounded by similar mechanisms. In the frontend BlockClient, we optimize the stack by moving the processing from kernel to user-space and then to hardware offloading in FPGA. In the backend BlockServer, we utilize high parallelism to achieve efficient throughput/IOPS control. As for space elasticity, EBS not only provides wide ranges of storage space (i.e., from 1 GiB to 64 TiB) but also supports features including flexible resizing and fast cloning.

We then move on to discuss threats to the availability of EBS, especially under large-scale deployment. We begin by categorizing three levels of failure events, including individual, regional, and global, that can lead to one, several, or all VDs in a cluster (temporarily) ceasing service. With VD segmentation and segment migration since EBS2, field diagnosis indicates that regional events become more frequent and an individual event can now easily cascade into regional or even global ones. Therefore, on the control plane, we developed a Federated BlockManager to organize the VDs into mini groups and use CentralManager for coordination. Additionally, on the data plane, we have built the logical failure domain to limit the destinations of segment migration.

In the third topic, we highlight the importance of offloading key control/data paths to hardware for acceleration. We use the offloading evolutions of both BlockClient and BlockServer as examples to discuss the tradeoffs between different options. Specifically, BlockClient started with FPGA offloading to accelerate storage/network virtualization. However, impacted by FPGA instability under large deployment (e.g., 22% of downtime caused by FPGA-related issues), our BlockClient dropped the FPGA-based approach and adopted the ASIC-based solution. Conversely, BlockServer, which also initially chose FPGA for speeding up EC/compression, opted for the many-core ARM CPU as the next stop due to the flexibility and cost requirement.

The final topic is organized as a series of "What If?" questions (§6). Through three Q&A, we explain why seemingly promising ideas, such as extending EBS1 with segmentation but without a log-structured design, eventually failed, and discuss the possibilities of alternative solutions (e.g., building EBS with open-source software). We end this paper with a short discussion of related work and a conclusion.

2 Architecture Evolution: A Shift of Focus

2.1 EBS1: An Initial Foray

EBS1 marked our first step into offering an elastic block store based on a disaggregated architecture. By placing the compute and storage in different clusters and connecting them via the datacenter network, this design offers flexibility in deployment, scaling, and evolution. Such philosophy has been widely adopted by many vendors, such as Microsoft Azure [25] and Google datacenters [30].

Compute end. Figure 2 provides a high-level overview of EBS1. A compute cluster comprises multiple servers where each server runs one BlockClient and can host several VMs. In addition, a VM can mount one or more VDs. Users can access the VD as a normal block device and the host server forwards the I/O requests to the storage clusters via the BlockClient.

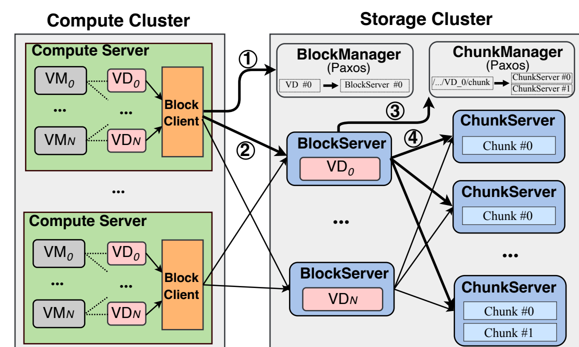


Figure 2: The system architecture of EBS1 (§2.1). ***VD:*** *Virtual Disk.* ***VM:*** *Virtual Machine. BlockManagers and ChunkManagers all run three-instance Paxos groups. Each VM can host multiple VDs.*

Storage end. EBS1 used a different fleet of dedicated servers for storage. First, we build the BlockManager (a set of three

nodes backed by Paxos) and a group of BlockServers. The BlockManager maintains VDs' metadata, such as the capacity and snapshot versions. The BlockServers handle I/O requests of multiple VDs assigned by the BlockManager. Note that the BlockManager can reassign a VD to another BlockServer during failover. Second, we further introduce a data abstraction, called *chunk*. The Logical Block Address (LBA) space of a user's VD is divided into a series of 64 MiB chunks. Similarly, the ChunkManager (a set of three nodes backed by Paxos) manages the chunks' metadata. The ChunkServer stores a 64 MiB chunk as a 64 MiB Ext4 file (called DataFile) and performs in-place updates to the chunk for write requests. Each chunk is three-way replicated on three different ChunkServers. For efficiency, we use thin provisioning—allocating space only when the user writes data to the VD.

Network. There are mainly two sets of network in `EBS1`. The frontend network connects the compute and storage clusters. The backend network inside the storage clusters connects the BlockServers to the ChunkServers. Both use Clos topology and rely on the 10 Gbps network with the kernel TCP/IP stack.

Data-flow. When a VM issues a new write request to its VD, BlockClient first contacts the BlockManager to locate the corresponding BlockServer for this request (① in Figure 2). Then, the BlockClient forwards the write request to the BlockServer (②). The BlockServer further asks the ChunkManager to determine the three ChunkServers (③) and persists the data accordingly (④). In practice, the BlockClient caches the VD-to-BlockServer mappings, and BlockServers cache chunk-to-ChunkServer mappings (i.e. skipping ① and ③).

Limitations. `EBS1`, released in 2012, has served over 1 million VDs and stored hundreds of PBs of data across hundreds of deployed clusters. Its straightforward and mature designs (e.g., in-place update and N-to-1 mapping between VDs and BlockServers)—while expediting development and deployment—limit the performance and efficiency. For example, to reduce the space overhead, we wish to use data compression and EC. However, compression non-deterministically alters the size of data which breaks the direct mapping of in-place updates. In addition, EC has a minimum size requirement (e.g., when the stripe unit is 4 KiB, EC(4,2) requires at least 16 KiB) and thus can result in significant write amplification, especially for small I/O requests. Another limitation is that, under the N-to-1 mapping, the performance of a VD is ultimately bounded by the performance of the corresponding BlockServer which can suffer from hotspot issues under burst workloads. In addition, with HDD and traditional kernel TCP/IP stack, we find it difficult to quantify and guarantee SLOs to users.

2.2 `EBS2`: Speedup with Space Efficiency

Overview. Figure 3 presents the high-level architecture of `EBS2`. The most significant change is that `EBS2` no longer directly handles the data persistence or manages the consensus protocol. Instead, it builds on top of a distributed storage system—named Pangu [35]—which provides append-only file semantics and distributed lock services (based on a customized Raft protocol). The BlockServers employ a log-structured design [38] and translate the VDs' write requests into Pangu append-only writes, thus enabling efficient data compression and EC during background garbage collection. We also split the VD address space into fixed-size segments, allowing one VD to be served by multiple BlockServers. Also, with segmentation, failover is no longer at the granularity of the whole VD but a segment—BlockManager migrates the impacted segment to another BlockServer. In addition, the BlockManager directly uses Pangu distributed lock service instead of Paxos for leader election.

As a result, we modified the I/O procedures as follows. After receiving a VD's request, BlockClient first retrieves the segment's address from BlockManager (① in Figure 3, which can be skipped by caching), and then forwards the I/O requests to the target BlockServer (②). BlockServer employs the Log-Structured Block Device (LSBD) Core to convert I/O requests into Pangu APIs and then calls an embedded Pangu client for persisting or fetching data (③). Note that, since `EBS2`, a BlockServer and a Pangu's ChunkServer, while co-located on the same physical server, are logically independent processes and rely on backend network for transferring data (i.e., not enforcing locality).

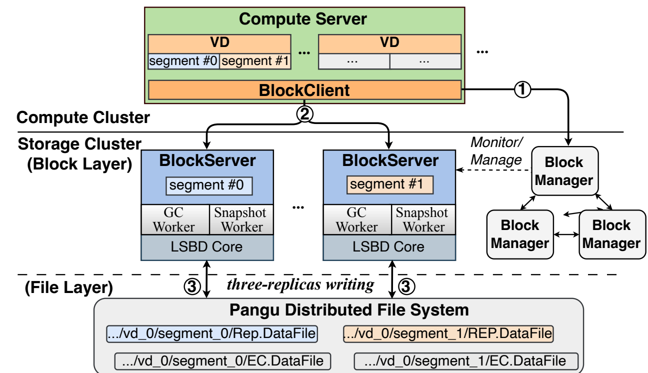


Figure 3: The overview of `EBS2`. ***LSBD***: *Log-Structured Block Device.* ***REP.DataFile***: *DataFile with three-way replication.* ***EC.DataFile***: *DataFile with EC(8,3) encoding.*

Disk segmentation. Figure 4 illustrates how `EBS2` partitions the VD's LBA into several 128 GiB segment groups each of which further comprises multiple 32 GiB segments. BlockServers in `EBS2` operate at the granularity of segments. Further, `EBS2` organizes the segment group as a series of data sectors and allocates them to the segments in a round-robin fashion. Finally, `EBS2` associates one segment with multiple *DataFiles* (512 MiB by default) to support concurrent writes. DataFile is essentially a Pangu file designed to persist a portion of a segment's data. These different levels of parallelism help `EBS2` alleviate the hotspot accessing in VDs.

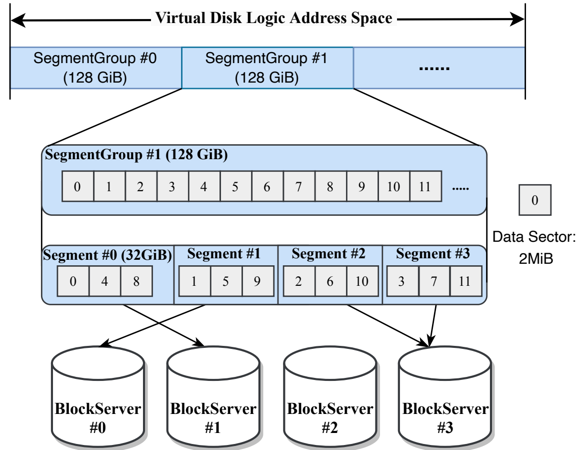


Figure 4: The Disk Segmentation Design of EBS2 (§2.2).

Log-structured Block Device. In EBS2, we developed a LSBD Core to support the append-only semantics of the underlying Pangu and thus split traffic into frontend (i.e., client I/Os) and backend (i.e., GC and compression). Figure 5 shows the frontend I/O flow including persisting users' data as a series of 4 KiB blocks and 64B metadata pairs (①), responding to users (②), recording updates in the transaction file (③), and updating the in-memory index map (④). The index map is essentially a log-structured merge tree (LSM-tree) to speed up the locating process by storing a mapping from VD's LBA to the corresponding DataFile ID, offset and length. The TxnFile accelerates the index map rebuild upon segment migrations. EBS can recover the in-memory index map by reading the latest I/Os' LBA-to-DataFile mappings from the TxnFile, without the need of tail scanning the data blocks in the DataFiles. Note that DataFile, TxnFile and the SSTables in the LSM-tree are all Pangu files.

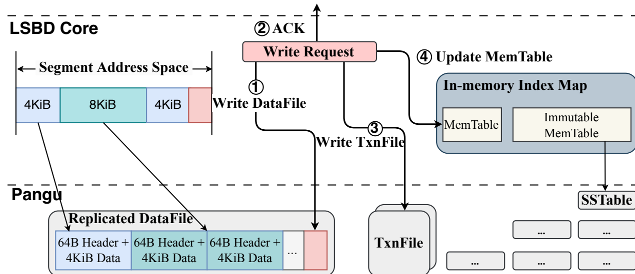


Figure 5: The data organization and persistence format of LSBD. *TxnFile: TransactionFile.*

GC with EC/Compression. EBS2 runs GC at the granularity of *DataFile* (see Figure 6). When stale data within DataFiles reaches the threshold, EBS2 initiates performing GC by collecting valid data from the dirtiest DataFiles under the same segment and combining them as new DataFiles. EBS2 finishes GC by updating the TxnFile and the in-memory index map. During GC, we also convert the replicated "REP.DataFiles" to space-efficient "EC.DataFiles" with (8,3) EC and LZ4 compression.

Given that compression can alter the size of data, we structured the EC.DataFile into three main components: a DataFile Header, a series of CompressedBlocks, and an Offset Table. The Compressed DataFile Header includes a magic number (marking the start of the DataFile), version and checksum. Each CompressedBlock contains CompressionHeader (CmpHdr) and CompressionBody (CmpBdy). The CmpHdr records the timestamp, the compression algorithm (LZ4 by default), the size of CmpBdy, and the checksum. CmpBdy contains the compressed data and metadata (i.e., 4 KiB + 64B before compression). At the end of the Compressed DataFile, we enclose the mapping between the original LBA in VD and the location in the Compressed DataFile as OffsetTable. When reading the compressed data, EBS2 first locates data by querying OffsetTable, then reads and decompresses the data.

We leveraged the opportunity of GC to perform the transformation of erasure coding and compression. If needed, EBS2 can schedule special types of GC tasks that preferably select "Rep.DataFiles" (replicated, non-compressed data) over existing "EC.DataFiles" (erasure-coded, compressed data), in order to make up more storage space for incoming writes.

The garbage percentage thresholds used to trigger GC in production vary, depending on the cluster storage usage and the workload. We deployed a set of optimizations for improving GC efficiency (e.g., placement based on inferring the block invalidation time [39]). In production, for the most stressed clusters, the write amplification due to GC (i.e., the number of bytes written by BlockServer and GCWorker over the number of bytes written by BlockServer) is less than 1.5.

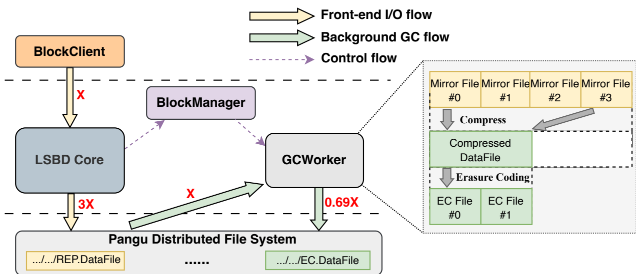


Figure 6: The Garbage Collection in EBS2.

BlockManager with higher availability. The integration of Pangu enhances the availability of EBS2's control plane. First, through the Pangu lock service, the BlockManager can continue serving clients even in the face of two out of three node failures. Second, EBS2 now stores the VDs' metadata in a persistent and replicated key-value store as Pangu files, while EBS1 stores the VDs' metadata in local disks where data loss could lead to an extended repair time.

Network. EBS2 uses a similar network setup as EBS1 except

for two fundamental differences. First, for the frontend network, we replace the kernel TCP with our user-space TCP implementation (called Luna [46]) over a 2×25 Gbps network. Luna achieves high performance (up to $3.5\times$ throughput improvement and 53% latency reduction) by leveraging a run-to-completion thread model and a zero-copy memory model. Second, for the backend network, we use a 2×25 Gbps RDMA network to meet the demanding SLAs [29]. Note that the two changes above only affect the data path. For the control path, we still use the kernel TCP (e.g., RPCs between BlockManagers and BlockServers).

Snapshot. The architectural changes in EBS2 also facilitate creating snapshots for VDs. With the out-of-place update, creating a snapshot in EBS2 no longer blocks foreground traffic. Instead, when receiving a snapshot request, BlockManager simply records a timestamp and asks the snapshot workers in BlockServers to upload the updates between the last snapshot timestamp and the latest one. As a result, generating a snapshot for 20GB of new data only takes around 30 seconds in EBS2, much less than the average 33 minutes in EBS1.

Other background I/O. Apart from GC, one important background task is data scrubbing, which performs periodical scanning to detect anomalies such as disk corruption and CPU silent data errors. To minimize the performance impacts, we have separated the scrubbing traffic from the GC tasks, capped the scrubbing traffic at 10 MiB/s (i.e., scanning all DataFiles every 15 days), and leveraged the heartbeat mechanism for monitoring the scrubbing progress.

Deployment. We released EBS2 in 2015 and subsequently scaled to more than 500 clusters for 2 million VDs. EBS2 could provide a virtual disk with an average write latency of 100 μs ($12\times$ reduction than EBS1), a maximum IOPS of 1 M ($50\times$ increase), and a maximum throughput of 4,000 MiB/s ($13\times$ increase) for the guest OS. In the field, the compression ratio of the LZ4 algorithm is between 43.3% $\sim$ 54.7%, and the average compression ratio is 50.1%. With Compress-EC in GC, EBS2 can reduce space usage from 3 to 0.69 replicas. Since un-GCed DataFiles are still stored with three-way replication, the average number of replicas in EBS2 is 1.29 in the field on average. For better management, we also reduced the cluster size from 700 servers in EBS1 to around 100 in EBS2.

Limitations. EBS2 successfully improves the space efficiency but incurred heavy traffic amplification. Compared with EBS1, EBS2 increases the overall traffic from 3 (i.e., from three-way replication) to 4.69 (i.e., 3 from the frontend plus 1.69 from the backend GC), yielding only 15.5% of the network bandwidth for serving the VDs' requests. To alleviate this issue, one promising solution is to adopt online EC/Compression, which means to directly store users' data in erasure-coded and compressed format.

The challenges are twofold. First, erasure coding requires the raw data blocks to be at least 16 KiB to achieve high compression and encoding efficiency. However, in the field, nearly 70% of write requests are smaller than 16 KiB. Moreover, EBS aims to deliver 100 μs write latency, and accumulating enough data in such a short interval can be difficult. For example, in order to perform compression and erasure coding for all user writes (i.e., accumulating 16 KiB data blocks within each 100 μs interval), a segment needs to have a write throughput over 160 MiB/s—surpassing 90% of segments in production. Simply padding zeroes can result in an even higher traffic/space amplification. Second, even with the latency-optimized LZ4 algorithm, compressing a 16 KiB-sized data block still requires 25 μs for CPUs, and such overhead escalates significantly for larger ones, rendering an unacceptable performance penalty for our service.

2.3 EBS3: Foreground EC/Compression

Overview. EBS3 achieves online EC/Compression by utilizing a Fusion Write Engine (FWE) to merge small writes and adopt an FPGA to offload the compression computations. Specifically, EBS3 first leverages FWE to accumulate writes from different segments of different VDs (i.e., step ① in Figure 7). FWE then combines these incoming writes as DataBlocks and sends them to the FPGA-based accelerator for data compression (②). EBS3 then calls Pangu to persist the compressed DataBlocks as JournalFiles with EC(4,2), namely ③. After persisting JournalFiles, EBS3 sends acks to the VD indicating the I/O completion (step ④). Then EBS3 copies the uncompressed data and preserves them within the BlockServer's memory by segment (i.e., SegmentCaches, step ⑤). When the data in a SegmentCache reaches the threshold (512 KiB by default), EBS3 compresses the data via the host CPU, appends them to the DataFile with EC(8,3), and updates the TxnFile and in-memory index map (step ⑥).

For read requests, EBS3 first queries the SegmentCaches in the BlockServers as they have the latest data. If not found, EBS3 would further read the in-memory index map (i.e., the LSM-tree). Note that JournalFiles are EC-ed with compressed data from various segments of different VDs. Directly fetching data from JournalFiles can result in severe read amplification (i.e., decompressing with heavy scanning). Therefore, JournalFile is write-only during runtime and only readable during failover to recover yet-to-be-dumped data upon crash.

Fusion Write Engine. Usually, when receiving a batch of small write requests, FWE waits until the total amount of data reaches a threshold (16 KiB by default) and then forms a DataBlock before sending it to the FPGA for compression. We set the waiting timeout as 8μs (i.e., the interval between NIC pollings). Moreover, we discover that insisting on merging smaller writes (e.g., 4 KiB) can result in higher 99th tail latency (i.e., 220%). Therefore, for clusters that have infrequent small writes (e.g., certain clusters, on average, with only 3.72% of the workload are 4 KiB writes), we do not aggregate 4 KiB writes but directly append them with the traditional three-way replication.

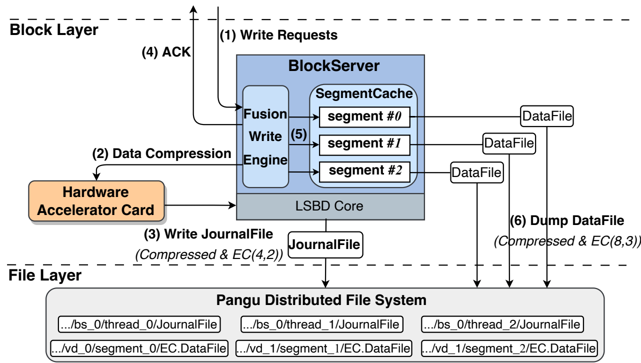


Figure 7: The architecture and I/O flow of EBS3.

FPGA-based compression offloading. We employ a customized FPGA to accelerate (de)compression, which includes a submission queue to buffer newly-formed DataBlocks and a completion queue to poll the results of hardware compression. We implement a scheduler inside the FPGA to split the DataBlocks as fixed-sized (e.g., 4 KiB) slices and employ multiple execution units to perform (de)compression tasks on these slices in parallel. To ensure data integrity, we place an end-to-end CRC check within the driver. After FPGA returns compressed data, EBS3 would immediately decompress the data and verify data integrity via CRC checking. Note this is a necessary overhead as, during failover, JournalFiles are the only data source.

Figure 8 shows the latency and maximum throughput of FPGA-offloading and CPU-only compression across different data block sizes, based on Silesia Compression Corpus [27]. The latency distribution of FPGA-offloading ranges from 29$\sim$65 μs. Notably, when the data block size is 16 KiB, the latency of FPGA-offloading reduces by 78% compared to CPU-only. Further, FPGA-offloading achieves a maximum throughput of 7.3 GiB/s, whereas CPU-only compression is only 3.5 GiB/s. As data block size increases, the FPGA-offloading solution leads to larger performance gains.

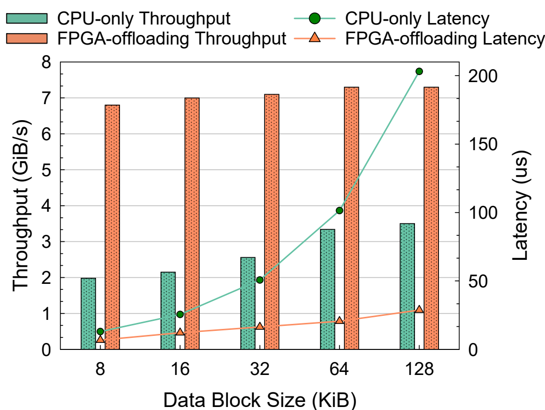
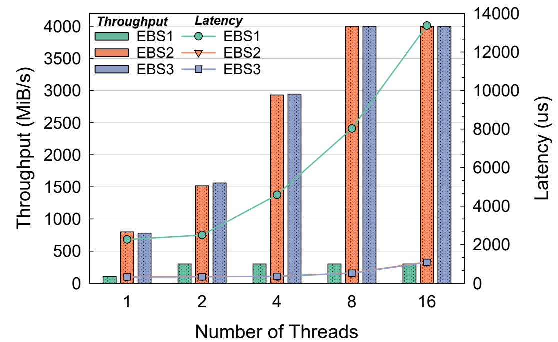


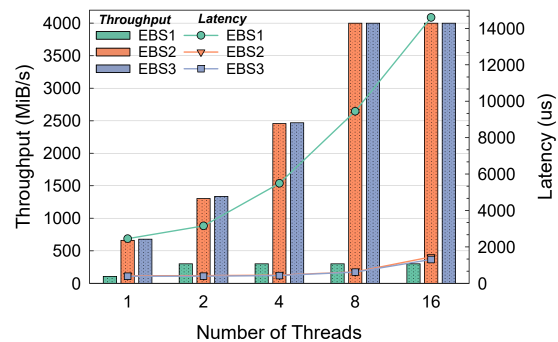
Figure 8: The compression performance comparison of FPGA-offloading and CPU-only with 8 cores compression based on Silesia Compression Corpus.

Network. EBS3 adopts higher linkspeed (i.e., 2×100 Gbps) network for both frontend and backend. In addition, we further developed Solar [36], a UDP-based transmission protocol. By leveraging the hardware offloading on our Data Processing Units (DPUs), Solar can pack each storage data block as a network packet, thereby achieving CPU/PCIe bypassing, easy receive-side buffer management and fast multi-path recovery.

Deployment. EBS3 has been deployed in over 100 storage clusters, serving more than 500K virtual disks since released in 2019. EBS3 offers comparable performance to EBS2. The incorporation of foreground EC/Compression in EBS3 enables all data to be stored immediately with high storage efficiency except in a few corner cases. As a result, the space efficiency (i.e., replica per data) in the field further drops from 1.29 in EBS2 to 0.77 in EBS3. In addition, the FPGA-based compression offloading can achieve 7.3 GiB/s throughput per card and the overhead ranges from from 29$\sim$65 μs. The overall traffic amplification drops from 4.69 in EBS2 to 1.59 in EBS3 (based on field statistics and numbers may slightly vary due to compression ratio differences across workloads).



(a) Throughput and Latency of Random **Write** on Thread-to-core Pinning



(b) Throughput and Latency of Random **Read** on Thread-to-core Pinning

Figure 9: Random Write/Read Latency of Each Generation EBS under Multiple Threads and 4 KiB-sized I/O. Thread-to-core pinning means that each thread occupies one CPU core exclusively.

2.4 Evaluation

To quantitatively demonstrate the improvement led by the architectural evolutions, we extensively evaluate the perfor-

mance of EBS1, EBS2 and EBS3 with microbenchmark (by stressing the VDs using FIO [21]) and application-based macrobenchmark (via RocksDB 6.2 with YCSB [26] and MySQL 8.0 with Sysbench [33]).

We evaluate the throughput and IOPS of the candidates by stressing random 4 KiB write and read. We also increase the number of threads (i.e., FIO jobs) from 1 to 16. Figure 9 shows the overall results. We can see that the throughput (i.e., bars) of EBS2 and EBS3 increases almost linearly before hitting the peak with 8 threads. With 8 threads, EBS2 and EBS3 can deliver 4,000 MiB/s throughput per VD (i.e., 1 M IOPS), which is 13× and 50× higher than the throughput and IOPS of EBS1. Figure 9 further depicts the latency variation with scaling of FIO jobs. We observe that the latency of EBS2 and EBS3 is similar and remains the same from 1 to 8 threads. Their latencies only experience a slight increase with 16 jobs due to delays caused by contention between threads after hitting throughput bottlenecks.

We use RocksDB with the default configuration. For MySQL, we use InnoDB and configure the user buffer size as 1 GiB, the page size as 16 KiB, the flushing method as direct I/O, the ramp-up time as 180 seconds and the execution time as 20 minutes. Figure 10 shows the results. Compared to EBS1, we observe 550% and 573% gains in throughput for insert and update—two write-dominated workloads—in YCSB, respectively. For read-dominated workloads, the throughput experiences an approximately 470% increase. Under oltp_insert workload in Sysbench, the throughput of EBS2 and EBS3 increase by 389% compared to EBS1. For the rest, the throughput increases by an average of 350%.

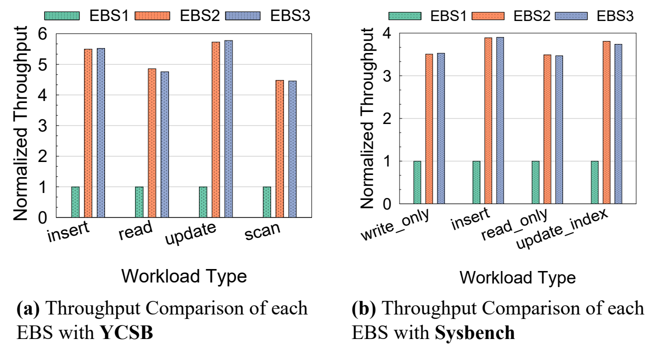
3 Elasticity: A Tale of Four Metrics

The capabilities (e.g., capacity and throughput) of a common block device (e.g., HDD and SSD) are usually bounded by the physical properties, such as encapsulation or interface. Backed by the cloud, EBS can provide VDs with much higher flexibility. In this section, we will share our experience of obtaining high elasticity, including pushing the upper bounds and achieving fine granularity.

3.1 Latency

The latency of a VD is determined by the architecture, namely the path a request has traveled. For example, the latency of an EBS2-backed VD is bounded by the latency of the two-hop network (from BlockClient to BlockServer and then to ChunkServer), the software stack processing (i.e., BlockClient, BlockServer and Pangu) and the SSD I/O. Hence, the elasticity of latency is inherently coarse-grained, namely the different levels of time overhead under various architectures (e.g., EBS2 and EBS3). Next, from the perspectives of average and tail latency, we further analyze the status quo.

Average latency. In Figure 11a, we measure the 8 KiB random read/write average latency breakdowns across different generations of EBS in their corresponding top 10% busiest

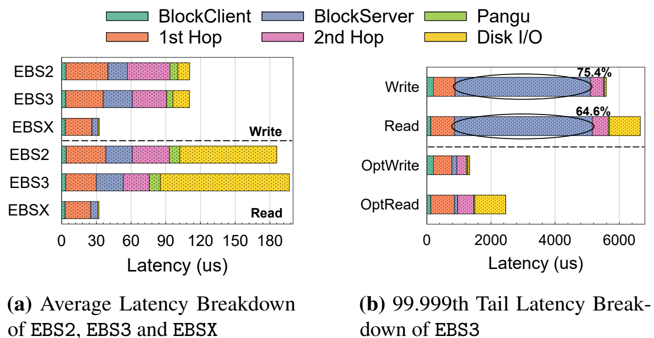


(a) Throughput Comparison of each EBS with **YCSB**

(b) Throughput Comparison of each EBS with **Sysbench**

Figure 10: Throughput Comparison (Normalized with EBS1).

production clusters. We choose to not include EBS1 in the comparison as it is no longer deployed and many of its hardware (e.g., HDD and 10 Gbps network) are obsolete. From the comparison, we first observe that the hardware processing—including 1st/2nd hop network (marked as orange and pink) and disk I/O (yellow)—accounts for the majority of the total latency in both EBS2 and EBS3. In addition, while EBS3 requires more time to process the data due to the frontend EC/compression, the reduced data volume in return spends less time traveling the network (i.e., lower 2nd hop latency in EBS3), yielding similar overall latency between EBS2 and EBS3. Third, the major difference between read and write lies in the disk I/O latency. Note that EBS2 is backed by TLC-based SSDs while EBS3 is backed by QLC-based SSDs.



(a) Average Latency Breakdown of EBS2, EBS3 and EBSX

(b) 99.999th Tail Latency Breakdown of EBS3

Figure 11: 8 KiB-Sized Avg. and Tail Latency Breakdown of EBS. ***1st hop:*** *network latency from compute to storage end.* ***2nd hop:*** *network latency from BlockServer to Pangu.*

Clearly, the key to improving average latency is reducing the hardware processing overhead. Therefore, we built EBSX, targeting latency-sensitive scenarios. EBSX installs the persistent memory (PMem) inside the BlockServers and directly stores the data in PMem with three-way replication. Compared to EBS2 and EBS3, EBSX skips the 2nd hop and drastically speeds up the disk I/O with PMem. Figure 11a shows that EBSX achieves 30 μs-level latency on both read and write. Note that, for space efficiency, data in PMem would be eventually flushed to Pangu and read statistics in Figure 11a is performance under cache hit (i.e., data in PMem).

Tail Latency. A user request may not be always served in time due to hardware failures [43, 44], misconfigurations [29, 36], software bugs [23, 32] or simply resource contentions [20]. Figure 11b presents the breakdowns of 99.999th percentile latency, a common threshold for defining the tail latency [28], among EBS3 clusters. We have collected millions of slow requests and calculated the average latency of each procedure. We observe that the BlockServer processing, such as non-IO RPC destruction in the IO thread, background periodic scrubbing and index compaction, accounts for the majority (75.4% for write and 64.6% for read) of the tail latency.

This observation may sound rather counter-intuitive as the hardware-related issues are usually blamed as the culprits [42, 45]. However, in EBS2 and EBS3, we have already incorporated a series of simple techniques to improve the quality of service of hardware processing. For example, network multi-path transport allows user requests to automatically shift traffic to other paths to avoid slow IO when suffering network abnormalities [36]. As another example, we employ a backup read/write strategy to trigger a retry to another location if the I/O takes longer than 99.9th percentile latency.

Our analysis indicates that the principal driver of the tail latency is the contention between the IO and the background tasks (e.g., segment status statistics and index compaction) in the BlockServers. In EBS2 and EBS3, the IO and the background tasks are executed on the same thread, leading to the IO hang-ups when the background tasks are triggered. To address this, we segregate the IO flow from other tasks and execute it on independent threads. With these enhancements, the 99.999th percentile write latency of EBS3 has been reduced to 1 ms, and the read latency reduced to 2.5 ms (i.e., OptWrite/Read in Figure 11b).

Summary. First, the elasticity of latency is coarse-grained—defined by the architectures, along with the hardware used (e.g., from EBS2 to EBSX). Second, optimizing hardware-induced latency is often straightforward. One can shorten the path (e.g., skip a network hop), use faster devices (e.g., PMem) or simply offset the risks with multi-path or retries. Third, tail latency by software stack has not received enough attention and may be regarded as noise. Our analysis suggests that under the high-speed network and fast SSDs, software-induced tail latency can be the dominant factor.

3.2 Throughput and IOPS

In the context of EBS, we often discuss the elasticity of throughput and IOPS together because the two metrics are often constrained by the same set of mechanisms. EBS achieves high elasticity in throughput/IOPS by optimizing two components on the key data path, BlockClient and BlockServer.

BlockClient. Every IO issued from the VD is first touched by the BlockClient. Therefore, the throughput and IOPS are bounded by the BlockClient's processing and forwarding capability. In EBS1, the BlockClient is implemented as a kernel module, and all IO requests are processed by the CPU. In

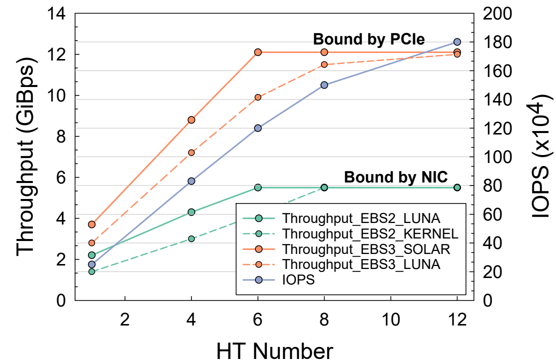


Figure 12: The maximum throughput and IOPS changes of BlockClient with different HT numbers.

EBS2, we move the IO processing to the user space by introducing a user-space TCP stack to handle the IO requests [46]. In EBS3, we further offload the IO processing to the hardware where a general-purpose FPGA, completely bypassing CPU, performs direct data move from VMs, data block CRC calculation, and packet transmission [36].

In Figure 12, we measure the maximum throughput and IOPS of BlockClient under different optimizations. We observe that EBS2 with the 2×25 Gbps network, throughput is constrained by network capabilities. For EBS3 with the 2x100G network, the bottleneck shifts to the PCIe bandwidth. Furthermore, as long as network bandwidth is available, IOPS increases with the number of hyper-threads (HTs).

BlockServer. Unlike BlockClient, once the requests reach the BlockServer, the throughput and IOPS are constrained by the levels of parallelism. Obviously, the more a request can be divided and served in parallel, the higher the throughput and the IOPS. Since EBS2, we have introduced three levels (i.e., SegmentGroup, Segment and Data Sector) of parallelism to enhance virtual disk performance.

Recall that the Data Sector size (initially 2 MiB) is configurable in the segmentation design. Reducing the Data Sector size allows virtual disks to scale one SegmentGroup across more BlockServers, thereby obtaining higher throughput/IOPS. In the field, we further decrease Data Sector size to 128 KiB and EBS2 (and EBS3) are able to deliver 1,000 IOPS for every GiB subscribed. Note that configuring an even smaller Data Sector size may backfire as the write requests can be too fragmented, thereby leading excessive number of sub-I/Os even for small writes and placing prohibitively high pressure on the first-hop network.

Base+Burst allocation. With high throughput/IOPS enabled by BlockClient and BlockServer optimizations, efficiently allocating throughput/IOPS to VDs is the next step. Unlike the coarse-grained elasticity in latency, users can subscribe throughput and IOPS of VD on demand without altering the capacity, which is called auto performance level (AutoPL). However, we discover that the throughput/IOPS in practice is often over-provisioned by users to handle the sporadic

workload bursts. For better resource efficiency, we have proposed the *Base+Burst* strategy based on the following techniques.

- *Priority-based congestion control.* We categorize IOs into baseIO and burstIO. BaseIO is pre-defined during virtual disk creation, while burstIO is allocated based on the available capability (i.e., not guaranteed). When a BlockServer is unable to meet all IO demands, it prioritizes processing the baseIO to ensure consistent latency. Currently, the maximum baseIO capacity of a VD is 50,000 IOPS, and the maximum burstIO capacity is 1 million IOPS.
- *Server-wise dynamic resource allocation.* Burst workloads can also place a heavy burden on BlockServer processing ability. Therefore, since EBS2, we devise the dynamic resource allocation to allow BlockServer to preempt resources (bandwidth, CPU cores and memory) from background tasks (e.g., GCWorker) to handle workload spikes.
- *Cluster-wise hot-spot mitigation.* To ensure enough headroom for burstIO, especially under concurrent bursts onto the same BlockServer, we use cluster-wise load-balancing to remove hot spots. Under such scenarios, BlockManager would aggressively check the traffic status and migrate segments more frequently between BlockServers.

Summary. The first lesson here is that the upper bound of a VD's throughput/IOPS is determined by the client (i.e., processing/forwarding ability) as the backend can easily scale with parallelism. In addition, the high throughput/IOPS is often desired but not always needed. Therefore, using a *Base+Burst* strategy to cope with workload spikes can be more economically beneficial for both users and vendors.

3.3 Capacity

Achieving elasticity in capacity is a fundamental requirement of a cloud block store service. In EBS, we have further included the following features.

- *Flexible space resizing.* The segmentation design enables EBS with seamless support for VD resizing (i.e., adding or removing SegmentGroups). EBS currently supports virtual disk sizes ranging from 1 GiB to 64 TiB.
- *Fast VD cloning.* One outstanding characteristic of serverless applications is a large volume of resources (e.g., VDs) needs to be allocated in a short time. To support this, EBS uses the *Hard Link* of Pangu files, which allows the cloning of multiple disks within a storage cluster via downloading a single snapshot. As a result, EBS2 enables the creation of up to 10,000 virtual disks (each 40 GiB) in 1 minute.

4 Availability: The Dark Side of Scaling

Availability has always been a priority of cloud services. In EBS, we especially focus on the *blast radius*—defined as the number of VDs experiencing unavailable services upon failures. Here, we categorize the blast radius as follows.

- *Global.* In the face of such an event, the service availability of an entire cluster is impacted. A simple example is an abnormally operating BlockManager causing the entire cluster to perform in an undesired fashion (e.g., a misconfiguration causing network congestion and subsequent retry storms). Note that EBS service runs on a per-cluster basis and we do not discuss datacenter-level failures here.
- *Regional.* For a regional event, we define it as a failure that incurs the component(s) to deny service for several VDs. For example, when a BlockServer crashes, the hosted VDs would experience an outage until the corresponding segments are migrated or all incoming I/Os are forwarded.
- *Individual.* When an individual event occurs, only one VD is influenced. Representative examples include an uncorrectable error inside the disk (and subsequently a read retry) and a software bug that leads to an unsuccessful and redirected write.

A straightforward solution to minimize the blast radius is setting smaller clusters. In EBS2 and EBS3, we have reduced the cluster size from 700 nodes (in EBS1) to around 100. The benefit is obvious since there are much fewer VDs influenced by a global event now. However, this approach, while straightforward and effective, would not alleviate regional and individual failure events.

Meanwhile, the *regional* events are likely to be more severe due to two trends. First, EBS2 introduces the segmentation to split one VD into 32 GiB segments and hence a VD in EBS2 (or EBS3) is supported by multiple BlockServers instead of one in EBS1. Moreover, the average capacity of VDs only slightly increases from EBS1 to EBS2 and EBS3 (e.g., 197 GiB to 220 GiB). Therefore, we can conclude that in EBS2 and EBS3, a BlockServer hosts much more VDs than EBS1. Consequently, when a BlockServer crashes, more VDs are going to be influenced.

Moreover, *individual* events can cascade into *regional* failures as the segments can be migrated since EBS2. For example, an internal incident occurred as tens of BlockServers in an EBS2 cluster kept crashing and rebooting, degrading the total cluster capacity and the I/O quality of thousands of VDs. In the beginning, a faulty segment crashes its BlockServer because of a buggy code logic—an individual event. Then, the control plane tries to migrate the segment to other BlockServers. However, as the client keeps retrying the failed requests, every BlockServer that loads the segment crashes as well, turning the individual event into a regional failure. This failure can easily grow to a cluster-wise outage in a short period if not manually intervened. Note that the cascading failures are not unique to EBS (e.g., cases in HBase [4–6]).

To adapt to the trends in regional and individual events, we further come up with techniques in both the control and data plane to improve the availability in EBS.

4.1 Control Plane: Federated BlockManager

In each cluster, the control plane of EBS2 (referred to as BlockManager in §2) initially consists of a group of three nodes that leverages the distributed lock service provided by Pangu for leader elections. The leader node in the group serves all the control plane requests to the corresponding cluster and persists any state changes related to VDs to a single metadata table stored in Pangu.

This setup presents two challenges. First, the single leader serves all the VDs in the cluster with one single server. As the VD's density grows, the chances and scale of its service disruption get higher. Second, the single metadata table hosts the metadata of VDs in the cluster. Once a part of the metadata table becomes corrupted, the BlockManager may not be able to load the metadata in memory, and cannot serve the VDs until the metadata table is repaired. In production, we have seen an increasing VD-per-cluster density over the past several years, urging us to solve both issues to provide high availability. Specifically, since the initial attempt to deploy a 100-node cluster in EBS2, the average number of VDs has increased from 20,000 to 100,000 per cluster. Correspondingly, the CPU and the memory consumption have increased by 4.1× and 4.5×, respectively.

Figure 13 illustrates the architecture of Federated BlockManager—our solution to the availability issue in the control plane. Each cluster now has multiple BlockManagers, with a CentralManager dedicated to managing the BlockManagers. Each BlockManager further manages hundreds of VD-level *partitions*, each of which corresponds to a metadata table that only stores the metadata of a small subset of VDs. The mappings from VDs to partitions are static; given a VD, we use hashing algorithms to compute its corresponding partition. The Client is not aware of the partition concept. For a given VD, it can query all the BlockManagers in the cluster to find out the BlockManager in charge, then send all its control plane requests to the BlockManager.

Note that instead of having three nodes (one leader and two standby), we now have only one node in each BlockManager. Upon the failure of a BlockManager, CentralManager redistributes its partitions to other BlockManagers, which then load the metadata tables of the partitions from Pangu into memory, and start to serve the VDs. Since the number of VDs in a partition is relatively small, the loading time is only several hundred milliseconds, without the need to have standby nodes continuously fetching the latest metadata updates for a fast leader switch. To ensure the availability of partition scheduling, the CentralManager consists of three nodes based on the lock service of Pangu.

Having multiple BlockManagers effectively reduces the blast radius of single leader service disruptions, since now each BlockManager only processes the requests of the subset of VDs in the partitions it manages. For the single table issue, instead of creating more BlockManagers, we adopt the partition design for two reasons. First, with partitions, we can make the number of VDs in a single metadata table small. For 100,000 VDs in a cluster, we need to have 1,000 tables to make the blast radius of metadata table failures smaller than 100, while it is not resource-efficient to create 1,000 BlockManagers in 100-node clusters. Second, the concept of multiple BlockManagers is mainly for distributing the workloads to multiple servers, so as to reduce the chances of service disruptions due to CPU and memory resource limits. Note that the CentralManager only manages BlockManager registrations and partition scheduling, and thus is less relevant to system availability.

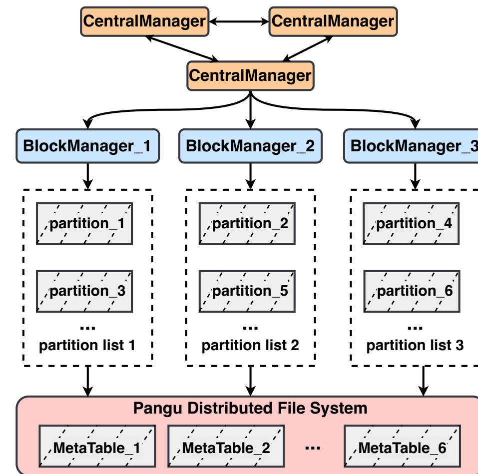


Figure 13: The architecture of Federated BlockManager.

Similar designs for blast radius reduction in the control plane of distributed storage systems can be found in the industry. HDFS Federation [1] is similar to Federated BlockManager, while it does not consider the metadata table failure mode. Each set of NameNode in HDFS Federation persists the metadata to its local disk, and all the requests to a set of NameNodes become unavailable when the data in the local disk is corrupted. AWS Physalia [24] deploys small units called cells, each of which consists of seven nodes deploying Paxos algorithms and serves a group of VDs. Differently, Federated BlockManager adopts a two-level VD management scheme, with each level emphasizing the blast radius of a single node and single table failures, respectively.

4.2 Data Plane: Logical Failure Domain

The data plane of EBS2 constitutes multiple BlockServers, each of which hosts thousands of segments and handles I/O requests of the segments. When a BlockServer crashes, the control plane migrates the segments to other BlockServers in the cluster, such that the I/O services can resume in other BlockServers. However, this mechanism indicates that failures can be cascading among BlockServers. Specifically, if the crash is caused by an error segment (e.g., recall the buggy code case in our production earlier), after the migration, the BlockServer shall resume the requests and crash again. Ironi-

cally, optimizing segment scheduling for faster recovery in this case can make more BlockServer crashes, even possibly leading to cluster-wise outages.

We have got several observations based on our deployment experiences. First, the failure typically originates from requests of a single VD or segment, and thus we need to monitor the status of segments instead of BlockServers. Second, the root causes of the failures are mostly due to software errors (e.g., bugs or misconfigurations) as hardware or mechanical failures are unlikely to travel along with the migrated segments. Software-induced failures can be time-consuming to pinpoint the culprit, let alone build automatic tools for recovering. Instead, a more practical way is to proactively reduce the impacts. Third, the cascading failures, if not intervened, can propagate quickly to the whole cluster. As a distributed service, it is acceptable to experience a few BlockServers shutdowns but a not cluster-wise outage.

Based on these observations, we design the logical failure domain. The core idea is to isolate any suspicious segments by grouping them into a small set of BlockServers, to avoid other BlockServers or even the entire cluster being impacted. We first associate each segment with a unique token bucket with a maximum capacity of three. Each migration to a new BlockServer consumes one token, and the token is refilled every 30 minutes. When the token bucket is empty, any subsequent migrations of the segment can only be selected among the three pre-designated BlockServers, referred to as its logical failure domain. The token bucket design does not limit the number of migrations but the range of migrations. When the segment is successfully loaded in a BlockServer and can readily serve I/O requests for several minutes, we lift the failure domain constraints.

The segment-level failure domain can effectively isolate an error segment. However, if there are multiple error segments (e.g., from one VD) or even VDs, the segment-level failure domain is not sufficient to prevent multiple cascading failures from happening at the same time. Our solution is to merge the failure domains into one. Specifically, when there exists more than one segment forming a failure domain, we pick the first failure domain as the global failure domain, so as to ensure there are at most three BlockServers isolated for the cascading failures, without degrading the cluster capacity. Any subsequent segment with an empty token bucket will share the same global failure domain. Recall that we associate each segment with three tokens. As a result, the chances of a normal segment accidentally sharing the same migration path with an error segment is small (note that a cluster is of 100-node size), which effectively reduces false negatives. As such, we can efficiently identify any error segments with small impacts on other VDs.

After deploying the logical failure domain, we have successfully defended our system against several potential outages due to migration-induced cascading failures. Evidence shows that some open-source storage systems also suffer from the same issue from time to time. For example, the HBase community reports several bugs [4–6] that show similar symptoms and lead to system outages. However, their treatments focus on solving the bugs and reducing the blast radius with physical isolation (similar to our first attempts), while the logical failure domain prevents the cascading failure from causing a cluster-wise outage once and for all.

4.3 Lessons Learned

First, with denser SSDs (e.g., QLC NANDs) becoming readily available and the boosting processing ability of CPU or other customized hardware (e.g., DPU), one can expect that a crashed node in a distributed storage system—not just `EBS`—can impact increasingly more users. As a result, regional failure events would be more frequent and/or severe. The key benefit of Federated Managers in this case is that it enjoys a smaller blast radius without losing the flexibility of a large-scale cluster.

Second, owning a forwarding layer between the users and the underlying service is popular among distributed services, for example, distributed key-value store [2] and cloud storage services [22, 25]. However, this also means the request or certain data structures can be redirected to other destinations upon failures, leading an individual event (e.g., bug or misconfiguration) to become regional or even global. We do not aim at proactively recovering or avoiding these failures because such events, like bugs or human errors, can be unpredictable. Instead, the logical failure domain works in a reactive fashion by confining the suspects among a few controlled servers and does not rely on manual intervention.

5 To Whom the `EBS` Offloads

Offloading the software stack to specialized hardware for better performance or achieving certain features (e.g., bare-metal servers) has been gaining momentum from both the cloud [10, 11] and hardware vendors [8, 12, 13]. Along the evolution of `EBS`, we have also leveraged FPGA for both frontend BlockClient (i.e., running the customized UDP-based protocol, Solar [36]) and backend BlockServer for accelerating compression/EC in `EBS3`. In the following subsections, we first demonstrate the motivations behind the two offloading. More importantly, we discuss why the two eventually both dropped FPGA-based solutions but went on different paths—ASIC for BlockClient and ARM CPU for BlockServer.

5.1 Offloading BlockClient

Since `EBS2`, the frontend BlockClient has become a bottleneck as the backend BlockServers can utilize segmentation for high throughput/IOPS. The BlockClient has been bounded by CPU-heavy tasks, including calculating CRC, encryption and performing per-I/O table lookups. Our stress test reveals it takes 4 CPU cores to saturate a 2×25 Gbps NIC in BlockClient. The newly emerged high-speed network would require a doubled or quadrupled number of cores. More im-

portantly, Elastic Computing Service (ECS, our VM service)—co-located with BlockClient—requires the servers to be bare-metal-ready (i.e., all CPU cores would be allocated to users). Therefore, offloading BlockClient processing to customized hardware is not only recommended but a must.

We initially built an FPGA-based solution for BlockClient and later decided to directly deploy this version in our production systems. This is because, at the time, directly applying the ASIC-based solution requires much longer development cycles. On the other hand, utilizing the ASIC-based approach is also not desirable due to higher power consumption and increased system complexity.

However, after several years of running in production, we discovered that FPGA is actually not the ideal candidate for BlockClient offloading. The major drawback is the instability. Specifically, 37% of data corruption incidents, as identified by CRC mismatches, are directly caused by FPGA-induced errors such as overheating, signal interference, and timing issues. This is because FPGAs are sensitive to environmental conditions and require precise timing, which can be disrupted by various factors like temperature fluctuations and electrical noise. Moreover, FPGA-related issues account for 22% of BlockClient's operational downtime. Typical reasons include hardware malfunctions, software bugs in the FPGA logic, and incompatibility with updated system components. Apart from reliability concerns, the frequency of FPGA is rather limited (e.g., around 200 MHz to 500 MHz), thereby limiting its potential for adapting to high-speed networks.

Therefore, we later move on to adopt the ASIC-based solution. The preliminary deployment of FPGA-based BlockClient considerably saves our time and effort for transitioning to ASIC (i.e., around 12 months between the release of FPGA and ASIC solutions). Compared to FPGA, ASIC-based offloading incurs approximately 5% of the CapEx and around 1/3 of the power consumption. Also, ASICs are optimized at the hardware level for specific tasks, allowing for more efficient use of resources and higher clock cycles. Another key enabler is that the main functionalities of BlockClient, including data movement, data calculation (e.g., CRC and encryption) and network packet processing, are usually stable. Hence, we do not need to periodically redesign the chips. After deployment, field statistics indicate that the failure rate of ASICs is an order of magnitude lower than that of FPGAs.

5.2 Offloading BlockServer

The goal of offloading BlockServer is to reduce costs while maintaining performance. EBS3 introduces data compression in the foreground to reduce traffic and space amplification. However, even with the latency-optimized LZ4 compression algorithm, the compression latency for 16 KiB-sized data blocks remains elevated at 25 μs (25.6% of total write latency) for software-based, and it escalates significantly with larger data blocks. Moreover, to achieve 4,000 MiB/s throughput, at least 8 CPU cores are required, leading to heightened resource contention and diminished performance. Therefore, offloading is necessary to avoid the penalty incurred by software-based compression.

While FPGA-based offloading demonstrates superior performance metrics (see Figure 8), the field deployment has exposed its limitations in terms of sustainability and cost-effectiveness. First, BlockServer faces similar instability FPGA issues. Over the past year, out of every 10,000 deployed production BlockServers, we have documented on average around 150 instances of compression offload failures by FPGA exceptions. Second, we are exploring optimizing compression algorithms tailored to data blocks with varying temperature profiles to minimize storage overhead. For example, using the ZSTD algorithm for separated cold data blocks can further achieve an average 17% space reduction. However, the resource constraints inherent to FPGAs preclude the dynamic adaptation to various compression algorithms. Finally, compared to the scale of BlockClients, the scale of BlockServers is still not large enough to amortize the escalating costs associated with FPGA development.

In light of these considerations, we are reorienting the target of offloading towards server CPUs. This shift is motivated by the advent of multi-core CPUs and specialized computational units integrated within them, which offer superior cost-efficiency while maintaining comparable performance metrics. Noteworthy examples include *Kunpeng 920* ARM CPU [41], and *Yitian 710* ARM CPU [9], all of which are equipped with dedicated units for compression acceleration. The test results show that the average LZ4 compression latency of *Yitian 710* is marginally higher by 1.3 μs in comparison to FPGA-based offload, while 16 ARM cores attain equivalent compression throughput.

Unlike BlockClient, we chose not to use ASIC for BlockServers because of two reasons. First, with no bare-metal requirements, there are no limitations on using CPU cores in BlockServers. Moreover, the BlockServer functionalities (a.k.a., operators) are in an ever-changing fashion. For example, the introduction of new compression and garbage collection algorithms. In this case, applying ASICs may require a complete overhaul from time to time, which can be prohibitively expensive even for the cloud-level scale.

5.3 Field Experience & Lessons

First, FPGA is undoubtedly the first choice for many hardware offloading scenarios due to its high flexibility and competitive performance. We, too, adopted FPGA in both BlockClient and BlockServer as proof-of-concept. However, the frequent errors and high CapEx made us realize that FPGA might not be an ideal acceleration option for large scale storage systems.

Second, ASIC and ARM are both suitable for the compute-to-storage architecture but in a different way. Compute end is cost-sensitive due to its massive scale and has stable operators, such as processing (e.g., encryption) and forwarding, matching the characteristics of ASIC. Storage backend can

have frequent upgrades (e.g., improving GC algorithms and optimizing host FTL for ZNS SSDs) and prioritize low interference between tasks. Therefore, the many-core ARM CPU becomes a proper choice.

6 What If?

Evolving EBS has been a long journey. Along the way, it is not surprising we have conceived or even tried out promising ideas that are only later to be proved as impractical. Here, we summarize such discussions as a series of "What If?".

Q1: What if the log-structured design was never adopted?

When developing EBS2, we initially tried to extend the EBS1 with segmentation but later dropped the idea due to high engineering effort. Note that this idea (i.e., in-place update with segmentation), if developed, can meet our requirements, including high performance and space efficiency, for EBS2.

However, this idea still falls short for EBS3. Foreground EC/compression necessitates that a storage system aggregates a sufficient amount of data before the persistence to achieve efficient data reductions. For example, EC(8,3) needs 32 KiB data for one stripe with 4 KiB stripe unit size, and 16 KiB compression units yield higher compression ratios. In §2.2 we have shown the domination of small writes (less than 16 KiB) combined with the need for low write latency prevent us from aggregating 16 KiB data for a single segment. The log-structured design can easily tackle this problem by allowing segments from different VDs to be merged together and flushed to a journal log.

We believe that Ceph [3] faces a similar issue due to the lack of a log-structured layer like BlockServer. To utilize EC in Ceph Block Device and CephFS, with FileStore, users need to set up a cache tier with write-back mode before an erasure-coded pool. With BlueStore, Ceph performs partial writes (i.e., read-modify-write) to an existing stripe without aggregating data, yielding additional network overhead, while EBS3 always writes full EC stripes to Pangu.

Q2: What if we built EBS with open-source software?

At first glance, one might assume that a cloud-level block store could be constructed by using a series of open-source softwares. For example, we can use HBase [2] (i.e., a log-structured distributed key-value store) and HDFS [7] (i.e., a distributed file system) to replace the block layer (i.e., BlockServer and BlockManager) and the file layer (i.e., Pangu).

However, the two designs are markedly different. EBS is a co-design of the block interface, the software, and the hardware. In the block layer, indexing is specifically tailored for the block service. For each segment, the key space is deterministic. Specifically, each segment represents an address space of a 32 GiB segment consisting of 4 KiB blocks, and thus the key space only includes 8 million numbers. This deterministic feature allows for efficient memory allocation for the indexing structure. To achieve low I/O average and tail latency, EBS deploys hardware offloading (i.e., offload compression algorithms to FPGA and ARM CPU) and customized network protocols [36], and decouples non-I/O activities from the critical I/O path, e.g., using individual GCWorker to perform garbage collection and moving index compaction procedures to background threads. In the file layer, Pangu is a high-performance storage system that builds dedicated user-space file systems for high-speed storage devices (i.e., NVMe SSDs), deploys high-speed networks (i.e., RDMA), and incorporates various hardware-offloading technologies [35].

Q3: What if Pangu and EBS were never separated?

The short answer is that such integration would have significantly hindered the development of EBS. Recall that in EBS1, the BlockManagers and BlockServers are integrated with the persistence layer (i.e., ChunkServers and ChunkManagers). This organization becomes increasingly unacceptable with the scaling of our engineering team. First, the interfaces (between the block and chunk layers) grew exceedingly complex—at one point there were nearly 10 sets of persistence APIs in EBS1—in order to support various functionalities and performance optimizations. Maintaining such a complex codebase undoubtedly slows down the development schedule. As an example, around that time, it could take up to 10 months for EBS to release a major upgrade due to delays by software bugs or incompatibility between components.

Decoupling the underlying persistence layer (i.e., Pangu) from EBS and adopting a unified log-structured interface have clearly facilitated the development and ease of communication. In addition, the independent block layer enables rapid segment creation and migration across multiple storage nodes, irrespective of data block locations. The separated architecture also localizes the impact radius of single-point failure to each respective layer. Moreover, this disaggregation also allows us to integrate emerging technologies (e.g., FPGA-based accelerator) and extend Pangu as a general-purpose DFS for other services (e.g., object store [17] and file store [15]).

7 Related Work & Conclusion

Cloud block store is a popular service provided by most cloud vendors [14, 18, 19, 25, 30]. In addition, academia has made great efforts in building and optimizing such a system, such as Salus [40], Ursa [34], Blizzard [37], and LSVD [31]. This paper differs from the above as it not only chronologically revisits the evolutions behind our EBS designs, but also provides a comprehensive summary of lessons we have obtained along the road, including on elasticity, availability, hardware offloads and the failed/alternative attempts.

Acknowledgments

The authors would like to thank our shepherd Prof. George Amvrosiadis and anonymous reviewers for their meticulous reviews, the Pangu and Kuafu team for their tremendous support and Yikang Xu for his contribution in developing EBS2. This research was partly supported by Alibaba ARF program and NSFC(62102424).

References

[1] Apache Hadoop 3.3.6 - HDFS federation. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/Federation.html.

[2] Apache Hbase. https://hbase.apache.org/.

[3] Erasure code – ceph documentation. https://docs.ceph.com/en/latest/rados/operations/erasure-code.

[4] HBASE-14598. https://issues.apache.org/jira/browse/HBASE-14598.

[5] HBASE-22072. https://issues.apache.org/jira/browse/HBASE-22072.

[6] HBASE-22862. https://issues.apache.org/jira/browse/HBASE-22862.

[7] HDFS architecture guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[8] The fungible DPU™: A new category of microprocessor for the data-centric era : Hot chips 2020. In *Proc. of IEEE HCS*, 2020.

[9] Alibaba Cloud unveils new server chips to optimize cloud computing services. https://www.alibabacloud.com/blog/598159?spm=a3c0i.23458820.2359477120.113.66117d3fm03t9b, 2021.

[10] AWS Nitro system. https://aws.amazon.com/ec2/nitro/, 2021.

[11] A detailed explanation about Alibaba Cloud CIPU. https://www.alibabacloud.com/blog/599183?spm=a3c0i.23458820.2359477120.3.76806e9bESi3SD, 2021.

[12] Intel Infrastructure Processing Unit. https://www.intel.com/content/www/us/en/products/details/network-io/ipu/e2000-asic.html, 2021.

[13] NVIDIA BlueField Data Processing Units. https://www.nvidia.com/en-us/networking/products/data-processing-unit/, 2021.

[14] About Google Persistent Disk. https://cloud.google.com/compute/docs/disks, 2023.

[15] Alibaba Cloud Apsara File Storage NAS. https://www.alibabacloud.com/help/en/nas, 2023.

[16] Alibaba Cloud Elastic Block Storage devices. https://www.alibabacloud.com/help/en/elastic-compute-service/latest/block-storage-overview-elastic-block-storage-devices, 2023.

[17] Alibaba Cloud Object Storage Service. https://www.alibabacloud.com/help/en/object-storage-service, 2023.

[18] Amazon Elastic Block Store. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AmazonEBS.html, 2023.

[19] Introduction to Azure managed disks. https://learn.microsoft.com/en-us/azure/virtual-machines/managed-disks-overview, 2023.

[20] M. Ajdari, W. Lee, P. Park, J. Kim, and J. Kim. FIDR: A scalable storage system for fine-grain inline data reduction with efficient memory handling. In *Proc. of IEEE/ACM MICRO*, 2019.

[21] J. Axboe. Flexible i/o tester. https://github.com/axboe/fio, 2017.

[22] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proc. of USENIX OSDI*, 2010.

[23] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, et al. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *Proc. of ACM SOSP*, 2021.

[24] M. Brooker, T. Chen, and F. Ping. Millions of tiny databases. In *Proc. of USENIX NSDI*, 2020.

[25] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proc. of ACM SOSP*, 2011.

[26] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of ACM SoCC*, 2010.

[27] S. Deorowicz. Silesia compression corpus. https://sun.aei.polsl.pl/~sdeor/index.php?page=silesia, 2014.

[28] P. DeSantis. Peter desantis keynote recap - AWS re:Invent 2022. https://caylent.com/blog/peter-de-santis-keynote-recap-aws-re-invent-2022, 2022.

[29] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, et al. When cloud storage meets RDMA. In *Proc. of USENIX NSDI*, 2021.

[30] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. of ACM SOSP*, 2003.

[31] M. H. Hajkazemi, V. Aschenbrenner, M. Abdi, E. U. Kaynar, A. Mossayebzadeh, O. Krieger, and P. Desnoyers. Beating the I/O bottleneck: a case for log-structured virtual disks. In *Proc. of ACM EuroSys*, 2022.

[32] L. Huang, M. Magnusson, A. B. Muralikrishna, S. Estyak, R. Isaacs, A. Aghayev, T. Zhu, and A. Charapko. Metastable failures in the wild. In *Proc. of USENIX OSDI*, 2022.

[33] A. Kopytov. Sysbench. https://github.com/akopytov/sysbench, 2021.

[34] 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 ACM EuroSys*, 2019.

[35] Q. Li, Q. Xiang, Y. Wang, H. Song, R. Wen, W. Yao, Y. Dong, S. Zhao, S. Huang, Z. Zhu, et al. More than capacity: Performance-oriented evolution of Pangu in Alibaba. In *Proc. of USENIX FAST*, 2023.

[36] R. Miao, L. Zhu, S. Ma, K. Qian, S. Zhuang, B. Li, S. Cheng, J. Gao, Y. Zhuang, P. Zhang, et al. From luna to solar: the evolutions of the compute-to-storage networks in Alibaba Cloud. In *Proc. of ACM SIGCOMM*, 2022.

[37] J. Mickens, E. B. Nightingale, J. Elson, D. Gehring, B. Fan, A. Kadav, V. Chidambaram, O. Khan, and K. Nareddy. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *Proc. of USENIX NSDI*, 2014.

[38] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proc. of ACM SOSP*, 1991.

[39] Q. Wang, J. Li, P. P. C. Lee, T. Ouyang, C. Shi, and L. Huang. Separating data via block invalidation time inference for write amplification reduction in Log-Structured storage. In *Proc. of USENIX FAST*, 2022.

[40] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin. Robustness in the salus scalable block store. In *Proc. of USENIX NSDI*, 2013.

[41] J. Xia, C. Cheng, X. Zhou, Y. Hu, and P. Chun. Kunpeng 920: The first 7-nm chiplet-based 64-core ARM SoC for cloud services. *Proc. of IEEE Micro*, 2021.

[42] J. Zhang, M. Kwon, D. Gouk, S. Koh, C. Lee, M. Alian, M. Chun, M. T. Kandemir, N. S. Kim, J. Kim, et al. FlashShare: Punching through server storage stack from kernel to firmware for Ultra-Low latency SSDs. In *Proc. of USENIX OSDI*, 2018.

[43] T. Zhang, J. Wang, X. Cheng, H. Xu, N. Yu, G. Huang, T. Zhang, D. He, F. Li, W. Cao, et al. FPGA-accelerated compactions for LSM-based key-value store. In *Proc. of USENIX FAST*, 2020.

[44] X. Zhang, X. Zheng, Z. Wang, H. Yang, Y. Shen, and X. Long. High-density multi-tenant bare-metal cloud. In *Proc. of ACM ASPLOS*, 2020.

[45] B. Zhu, Y. Chen, Q. Wang, Y. Lu, and J. Shu. Octopus+: An RDMA-enabled distributed persistent memory file system. *ACM Trans. on Storage*, 17(3):1–25, 2021.

[46] L. Zhu, Y. Shen, E. Xu, B. Shi, T. Fu, S. Ma, S. Chen, Z. Wang, H. Wu, X. Liao, et al. Deploying user-space TCP at cloud scale with LUNA. In *Proc. of USENIX ATC*, 2023.

ELECT: Enabling Erasure Coding Tiering for LSM-tree-based Storage

Yanjing Ren[1], Yuanming Ren[1], Xiaolu Li[2], Yuchong Hu[2], Jingwei Li[3]*, and Patrick P. C. Lee[1]
[1]*The Chinese University of Hong Kong* [2]*Huazhong University of Science and Technology*
[3]*University of Electronic Science and Technology of China*

Abstract

Given the skewed nature of practical key-value (KV) storage workloads, distributed KV stores can adopt a tiered approach to support fast data access in a hot tier and persistent storage in a cold tier. To provide data availability guarantees for the hot tier, existing distributed KV stores often rely on replication and incur prohibitively high redundancy overhead. Erasure coding provides a low-cost redundancy alternative, but incurs high access performance overhead. We present ELECT, a distributed KV store that enables erasure coding tiering based on the log-structured merge tree (LSM-tree), by adopting a hybrid redundancy approach that carefully combines replication and erasure coding with respect to the LSM-tree layout. ELECT incorporates hotness awareness and selectively converts data from replication to erasure coding in the hot tier and offloads data from the hot tier to the cold tier. It also provides a tunable approach to balance the trade-off between storage savings and access performance through a single user-configurable parameter. We implemented ELECT atop Cassandra, which is replication-based. Experiments on Alibaba Cloud show that ELECT achieves significant storage savings in the hot tier, while maintaining high performance and data availability guarantees, compared with Cassandra.

1 Introduction

Storage tiering provides a storage paradigm for balancing the trade-off between access performance and storage persistence in large-scale storage. In particular, for distributed key-value (KV) storage, practical KV workloads are known to have skewed access patterns [6, 9, 16, 62], in which a small fraction of KV pairs are frequently accessed (i.e., hot) and the remaining large fraction of KV pairs are rarely accessed (i.e., cold). Thus, it is natural for distributed KV stores to adopt storage tiering, in which a *hot tier* provides fast data access for hot KV pairs, while a *cold tier* provides persistent storage with less-demanding performance requirements for cold KV pairs.

A primary use case for storage tiering is *edge-cloud storage*. Our motivation is that Internet-of-things (IoT) applications are forecast to generate over 79.4 ZB of data in the wild by 2025 [51]. Since the cloud access performance is bottlenecked by the constrained Internet bandwidth, IoT applications are often coupled with the edge computing paradigm, in which edge nodes, the lightweight instances (e.g., micro-datacenters) provisioned with limited computation and storage resources, are deployed in close proximity to IoT devices [52, 53]. From a storage perspective, we can deploy a distributed KV store as the high-performance hot tier in the edge, while the cloud forms the persistent cold tier. Such an edge-cloud storage architecture has been used in virtualized network functions [40], multi-tier computing [42], multimedia management [21], etc. In addition to edge-cloud storage, storage tiering is also applicable to other storage architectures, such as content-delivery networks and cloud block storage (e.g., the combination of Amazon's Elastic Block Store (EBS) [1] and Simple Storage Service (S3) [2]).

Providing data availability guarantees for hot-tier storage is necessary, especially when the hot tier is deployed in distributed storage environments where failures are prevalent [24]. For example, in edge-cloud storage, edge nodes have limited storage resources and are also prone to failures [53]. While the cloud provides abundant persistent storage resources, reconstructing any lost data for failed edge nodes from the cloud is inefficient due to the high edge-cloud latencies [12, 67]. Thus, to ensure data availability against edge node failures, the edge can introduce storage redundancy, so that any lost data in the edge can be directly reconstructed through the redundant data from other available edge nodes, without the need for retrieving data from the cloud for reconstruction. However, modern distributed KV stores [18, 22, 34, 46] are commonly designed for cloud data centers with sufficient resources, and adopt *replication* to distribute exact redundant copies for individual KV pairs across multiple nodes to provide fault tolerance against node failures. Replication multiplies storage overhead, which is prohibitive for resource-constrained edge nodes, or generally, the high-performance hot tier with limited storage resources.

Erasure coding provides a low-cost redundancy alternative to achieve data availability with much lower storage overhead compared with replication (see §2.3 for details). It has been extensively studied in the literature, especially for distributed KV storage in data centers (§7). However, there exists a fundamental storage-performance trade-off for replication and erasure coding: replication incurs high storage overhead, yet it supports not only load balancing of reads across redundant copies, but also simple reconstruction of any lost data from another available redundant copy; in contrast, erasure coding significantly reduces storage overhead, but it does not keep

*Jingwei Li is the corresponding author.

redundant copies for load balancing and is known to incur higher bandwidth and I/Os in reconstructing lost data when failures happen [19, 26]. It is thus critical to mitigate the storage overhead, while maintaining high access performance as if replication were used, in the hot tier.

We present ELECT, a distributed KV store that enables erasure coding tiering. ELECT builds on the log-structured merge tree (LSM-tree) [45]. Given the skewed nature of practical KV storage workloads [6, 9, 16, 62], ELECT extends the LSM-tree with a *hybrid redundancy* approach by storing limited amounts of hot KV pairs with replication in the hot tier for high access performance, while still achieving significant storage savings by storing large amounts of cold KV pairs with erasure coding in the hot tier. In addition, it can offload cold KV pairs from the hot tier to the cold tier to further alleviate the storage overhead in the hot tier.

Enabling hybrid redundancy in ELECT, however, is non-trivial. ELECT should decide *how* (e.g., at what granularities), *when* (e.g., on or off the write path), and *what* (e.g., differentiating hot and cold KV pairs) to convert replicated KV pairs into erasure-coded KV pairs. Most importantly, ELECT should provide a mechanism to balance the trade-off between storage savings and access performance; such a mechanism should be adaptive to various user requirements. Note that erasure coding has also been proposed for caching [49] and content delivery networks [61] in the context of storage tiering. The novelty of ELECT lies in the careful combination of replication and erasure coding for LSM-tree-based storage with several new design techniques (see §7 for details).

Our contributions are summarized as follows.

- We design ELECT to make a case for enabling erasure coding tiering for distributed KV storage. ELECT has several design features: (i) a redundancy transitioning approach for the conversion from replication to erasure coding based on the LSM-tree; (ii) a hotness-aware approach for both redundancy transitioning and the data offloading from the hot tier to the cold tier; and (iii) a tunable approach, with only a single user-specified parameter based on a storage saving target for simple deployment, for configuring how much data to be erasure-coded and offloaded.
- We implemented ELECT atop Cassandra v4.1.0 [3]. Cassandra [34] is a distributed KV store that uses consistent hashing [31] for data partitioning (§2.1) and the LSM-tree for internal storage management (§2.2). We choose Cassandra due to its decentralized, high-performance, and fault-tolerant nature. Cassandra supports only replication, and ELECT extends Cassandra with erasure coding tiering.
- We conduct experiments in an edge-cloud setting on Alibaba Cloud [38]. Compared with (replication-based) Cassandra, ELECT achieves 56.1% edge storage savings, with similar performance in normal read/write operations.

We now open-source our ELECT prototype at **https://github.com/adslabcuhk/elect**.

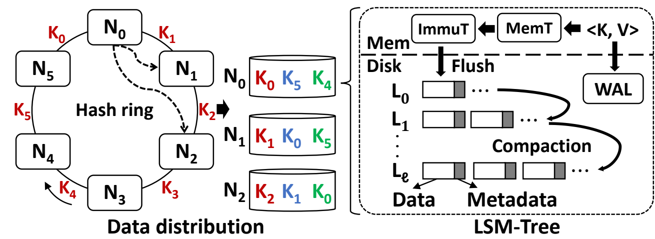


Figure 1: Triple replication in a distributed KV store, in which each node uses an LSM-tree for internal storage management.

2 Background

2.1 Distributed KV Stores

Modern distributed KV stores partition KV pairs across multiple nodes by either consistent-hashing-based [31] distribution [17, 18, 22, 34] or range-based distribution [5, 11, 46, 57]. Consistent hashing arranges nodes in a hash ring, in which each node is associated with a range of the hash ring and stores the KV pairs whose keys are hashed to the range. For load balancing, each node can be further associated with multiple virtual nodes that are associated with different ranges of the hash ring. In contrast, range-based distribution divides the entire key space into non-overlapping ranges, in which each node stores the KV pairs in one of the ranges. ELECT builds on Cassandra [34], which uses consistent hashing, yet its design is compatible with both distribution approaches.

Replication is commonly used in distributed KV stores for fault tolerance [5,11,18,22,34,46,57]. Given a replication factor R, replication distributes R copies of each KV pair across a set of nodes, which collectively form a *replication group*. Suppose that there are M nodes (denoted by $N_0, N_1, \cdots, N_{R-1}$) arranged in a clockwise direction of a hash ring. Each node N_i is associated with a non-overlapping key range K_i, where $0 \leq i \leq R-1$, that corresponds to the keys that precede N_i in the hash ring. Suppose that a KV pair is mapped to N_i. The first copy of the KV pair (called the *primary replica*) is stored in N_i, and the $R-1$ additional copies (called the *secondary replicas*) are stored in the following nodes along the clockwise direction of the hash ring (i.e., $N_{i+1 \bmod M}$, $N_{i+2 \bmod M}$, $\cdots$, $N_{i+R-1 \bmod M}$). Thus, each node N_i now manages the primary replicas of its associated key range as well as the secondary replicas of the associated key ranges of the $R-1$ preceding nodes in the anti-clockwise direction of the hash ring. Figure 1 shows an example of triple replication (i.e., $R=3$) and $M=6$ nodes. For example, N_0 stores the KV pairs for K_0, K_5, and K_4.

2.2 Log-Structured Merge Trees (LSM-Trees)

Each node in Cassandra [34] manages its internal storage based on an LSM-tree [45], as also used in state-of-the-art local [9] and other distributed [5, 11, 34, 46] KV stores. An LSM-tree is a data structure designed for efficient writes (i.e., `Put` requests), reads (i.e., `Get` requests), and scans (i.e., `Scan` requests). Figure 1 shows how an LSM-tree is deployed in

a distributed KV store. An LSM-tree organizes KV pairs in immutable fixed-size files, called *SSTables*, across $\ell+1$ levels, denoted by $L_0, L_1, \cdots, L_\ell$; L_0 is the lowest level and L_ℓ is the highest level. Each SSTable stores multiple KV pairs in a *sorted* manner in units of *data blocks* of size several KiB each. To support fast reads, each SSTable maintains an *index block* to track the key ranges and offsets of all data blocks as well as a *Bloom filter* [7] to track its currently stored keys with a small false positive rate. We refer to the data blocks as the *data component*, and collectively refer to the index block, Bloom filter, and SSTable metadata as the *metadata component*, for an SSTable. Inside the LSM-tree, the number of SSTables in each level increases from the lower to higher levels, while the KV pairs of an SSTable do not overlap with those of other SSTables in the same level except L_0 (note that some advanced LSM-trees may have overlapping KV pairs across SSTables in the same level [48]).

Writes and reads of KV pairs are issued to an LSM-tree as follows. Each write appends a newly written KV pair to an on-disk *write-ahead log (WAL)* for crash consistency and then inserts it into a mutable in-memory structure called a *MemTable*. When the MemTable is full, it is turned into an *Immutable MemTable*, which is then flushed to the lowest level L_0 as an SSTable. Each level has a capacity limit, increasing from lower to higher levels. When a lower level reaches its capacity limit, it triggers *compaction* to merge the KV pairs in the lower level into its next higher level. Specifically, a compaction operation selects an SSTable in the lower level, reads all SSTables in the higher level that have overlapping key ranges with the selected SSTable, sorts all the latest KV pairs (while all stale KV pairs are discarded), and re-generates and stores the non-overlapping SSTables in the higher level. On the other hand, each read for a key searches the MemTable and then the SSTables from L_0 to L_ℓ. It returns the KV pair if the key is found, or null otherwise.

2.3 Erasure Coding

Erasure coding provides low-redundancy fault tolerance for distributed storage. In this work, we focus on Reed-Solomon (RS) codes [50], configured by two parameters n and k (where $k < n$), as our erasure code construction. We choose RS codes for three reasons: (i) they support general coding parameters n and k (provided $k < n$); (ii) they have the minimum storage redundancy for tolerating against any $n-k$ node failures (a.k.a. the Maximum Distance Separable (MDS) property); and (iii) they have been popularly deployed in production [24, 43].

An (n,k) RS code encodes k original (uncoded) fixed-size *data chunks* into $n-k$ (coded) *parity chunks* of the same size, and the collection of n data and parity chunks forms a *coding group*; the (n,k) RS code considered here is *systematic*, meaning that the coding group keeps the k data chunks. It ensures that any k out of the n chunks of a coding group can reconstruct all k original data chunks. Large-scale storage systems comprise multiple coding groups that are independently encoded/decoded, and the n chunks of each coding group are distributed across n nodes to tolerate any $n-k$ node failures with $n/k\times$ storage overhead. Compared with replication, RS codes incur much lower storage overhead with higher fault tolerance; for example, Facebook [43] uses the $(14,10)$ RS code for four-node fault tolerance with $1.4\times$ storage overhead only, while traditional triple replication [25] only provides two-node fault tolerance and incurs $3\times$ storage overhead.

Erasure coding is known to have reconstruction penalty. For example, for any lost chunk, an (n,k) RS code needs to retrieve k available chunks from other alive nodes in the same coding group so as to decode the lost chunk. Reconstruction is common in practice due to the prevalence of failures [24, 26, 43], and there are two major reconstruction operations: degraded reads (i.e., reads issued to lost chunks) and full-node recovery (i.e., all data stored in a node is lost).

There are code constructions that reduce the reconstruction bandwidth of RS codes (e.g., regenerating codes [19] and locally repairable codes [26]). However, they still retrieve more data for reconstruction than the amount of lost data, and the trade-off between storage savings and reconstruction bandwidth in erasure coding is fundamental [19].

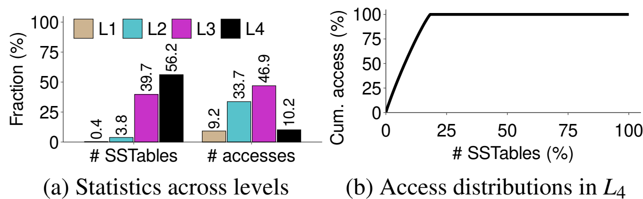
3 Design Considerations

Before we present the design of ELECT, we pose five design questions that need to be addressed.

Q1: At what granularity should KV pairs be encoded? In the context of KV stores, there are two approaches to encode KV pairs at different granularities: (i) *self-encoding* [8, 32, 33, 44], which splits a KV pair into k fixed-size data chunks for encoding, and (ii) *cross-encoding* [14, 37, 65, 66], which aggregates multiple KV pairs into individual data chunks and performs encoding on each group of k different data chunks. Self-encoding improves the parallelism of data access, but incurs significant metadata overhead for indexing all chunks of individual KV pairs [65], especially when KV services are dominated by small KV pairs [9]. In contrast, cross-encoding mitigates such metadata overhead, but the degraded read to a KV pair during a node failure needs to retrieve k surviving chunks of the same coding group for reconstruction, thereby leading to amplified I/Os and bandwidth.

ELECT opts for cross-encoding to reduce the metadata overhead; if we only encode cold KV pairs that are rarely accessed (see Q3 below), the degraded read overhead should be limited. Also, since LSM-trees organize KV pairs in units of SSTables, ELECT opts for cross-encoding across multiple SSTables (i.e., each SSTable is treated as a chunk) to align with the LSM-tree-based storage management.

Q2: Should erasure coding be performed on or off the write path? Erasure coding for KV pairs can be performed *inline* [8, 20, 44], in which KV pairs are encoded on the write path, or *offline* [23, 36, 58], in which KV pairs are first written and later encoded in the background. Offline encoding has the flexibility of first storing hot KV pairs with replication



(a) Statistics across levels (b) Access distributions in L_4

Figure 2: Storage and access patterns in Cassandra.

for high access performance (see Q3 below). Thus, ELECT opts for offline encoding, in which KV pairs are first written with replication, and later performs erasure coding (across SSTables) in the background.

Q3: How should skewed access patterns be addressed? Practical KV workloads have skewed access patterns [6,9,16, 62], in which few KV pairs are frequently accessed (hot) and the majority of KV pairs are rarely accessed (cold). ELECT opts to apply erasure coding to cold SSTables to mitigate the degraded read overhead in cross-encoding (see Q1), while storing hot SSTables with replication for high-performance accesses with limited additional storage overhead. The idea of applying replication for hot data and erasure coding for cold data has been studied in prior studies [23,36,58], yet they target different deployment environments and how to adapt this idea into LSM-tree-based KV stores remains unexplored.

We motivate our design by examining the storage and access patterns of SSTables in different LSM-tree levels by generating realistic KV workloads using the benchmarking tool YCSB [16], which has been extensively used for KV storage evaluation in the literature. Specifically, we load 100 M 1-KiB KV pairs with a key size of 24 bytes and a value size of 1000 bytes into Cassandra (v4.1.0) via YCSB in our testbed (see §6.1 for testbed details). Also, using the `nodetool` command in Cassandra, we flush the MemTable of each LSM-tree to disk and force the compaction on all SSTables to keep all nodes in a stable state. We find that the last level (i.e., the highest level) is L_4, while L_0 is empty as the SSTables originally in L_0 are merged to L_1 after the forced compaction. We then issue 10 M reads to the stored KV pairs, where the keys are accessed under the Zipf distribution with a Zipfian constant of 0.99 (default in YCSB). Note that we set the replication factor as one to mitigate the impact of replication, and disable the key cache and row cache in Cassandra to have all KV pairs read from on-disk SSTables.

Figure 2(a) shows the distributions of numbers of SSTables and accesses to SSTables in each level. The intermediate levels L_2 and L_3 have high read frequencies. However, L_4 stores the most SSTables (56.2% of all SSTables), but only accounts for 10.2% of accesses. This motivates us to perform erasure coding only for the SSTables in the last level (e.g., L_4 in this example) and replication for the SSTables in the lower levels, so that we still achieve significant storage savings and limit the degraded read overhead caused by erasure coding.

Figure 2(b) further shows the cumulative distribution of access frequencies versus the SSTables in L_4. Only 18.2% of SSTables in L_4 are accessed. This suggests that we can apply erasure coding in a more fine-grained manner by selecting only the SSTables that are rarely accessed for erasure coding (i.e., with negligible degraded read overhead).

Q4: How should the access overhead in the cold tier be mitigated? It is expected that the cold tier has worse access performance than the hot tier. For example, in edge-cloud storage, while the cloud provides much more abundant storage resources than the edge, it is also limited by the high edge-cloud latency over the Internet (e.g., 30 ms for client-to-cloud communication versus 5 ms for client-to-edge communication [12,67]). ELECT should selectively offload data that is rarely accessed from the hot tier to the cold tier, so as to avoid frequently retrieving the data back from the cold tier.

Q5: How should ELECT address the trade-off between storage savings and access performance? Both redundancy transitioning from replication to erasure coding and the data offloading from the hot tier to the cold tier in essence trade access performance for storage savings. ELECT should provide a tunable mechanism that allows users to balance the trade-off depending on their requirements.

4 ELECT Design

ELECT extends Cassandra [34], which uses replication for fault tolerance, with erasure coding tiering. It is deployed across multiple nodes in the hot tier and is backed by the cold tier with persistent storage. It proposes several design elements to address the questions in §3.

- *LSM-tree-based redundancy transitioning* (§4.1). ELECT applies cross-encoding (see Q1) across SSTables in an offline manner (see Q2), by converting SSTables from replication into erasure coding (called *redundancy transitioning*). It decouples the replicas originating from different nodes into multiple LSM-trees, such that it applies cross-encoding to the primary replicas and removes the secondary replicas after encoding. One subtlety is that it should maintain the correctness of redundancy transitioning even under LSM-tree compaction, which changes the content of SSTables.
- *Hotness awareness* (§4.2). ELECT applies cross-encoding to only SSTables in the last LSM-tree level (see Q3). It also offloads SSTables that tend to be rarely accessed from the hot tier to the cold tier to mitigate the access overhead in the cold tier (see Q4).
- *Balancing storage-performance trade-off* (§4.3). ELECT provides a user-configurable parameter, namely the storage saving target, to balance the trade-off between storage savings and access performance (see Q5).

4.1 LSM-tree-based Redundancy Transitioning

ELECT decomposes redundancy transitioning into four steps: LSM-tree management (§4.1.1), parity node selection (§4.1.2), cross-SSTable encoding (§4.1.3), and secondary replica removal (§4.1.4). Figure 3 shows the overall redundancy transitioning workflow in ELECT.

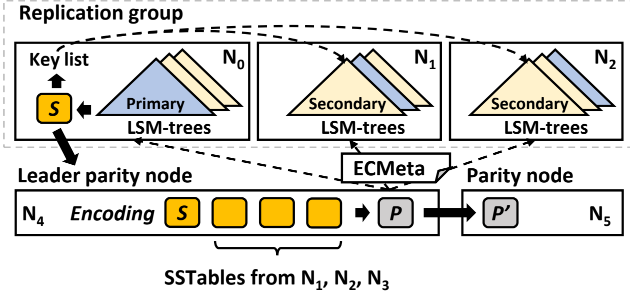


Figure 3: LSM-tree-based redundancy transitioning in ELECT for a coding group of (6,4) RS coding, in which N_0, N_1, N_2, N_3 are data nodes, while N_4 (leader parity node) and N_5 are parity nodes.

4.1.1 LSM-tree Management

Decoupled replication management. In Cassandra, all replicas stored in a node are managed in a single LSM-tree. To facilitate cross-SSTable encoding across nodes and subsequent removals of replicas, ELECT borrows the idea of decoupled replication management [57,68] (originally designed for reducing I/O amplification) by separating the replicas into multiple LSM-trees in each node. Recall that for a replication factor R, each node maintains the primary replicas originating from the node itself and the secondary replicas originating from the $R-1$ preceding nodes in the hash ring (§2.1). ELECT now lets each node maintain R LSM-trees, comprising one *primary LSM-tree* for the primary replicas and $R-1$ *secondary LSM-trees* for the $R-1$ respective sets of secondary replicas. For example, in Figure 1 with $R=3$, N_0 writes the primary replicas in key range K_0 to the primary LSM-tree and the secondary replicas in key ranges K_5 and K_4 to two other secondary LSM-trees.

LSM-tree level generation. The original LSM-tree creates a new level L_ℓ when the current last level $L_{\ell-1}$ is full. However, depending on the current storage usage, the last level L_ℓ may only contain a small number of SSTables. This compromises the effectiveness of ELECT, which applies erasure coding only to the SSTables in the last level. To address this issue, ELECT modifies the current LSM-tree design and creates the new level L_ℓ in the LSM-tree only when the size of the current last level $L_{\ell-1}$ reaches the capacity limit of the next level L_ℓ. For example, the LSM-tree in Cassandra currently sets the size limits of $L_{\ell-1}$ and L_ℓ as T and $10T$, respectively, where T is some capacity limit and the default capacity difference across adjacent levels is 10×. ELECT now keeps adding SSTables to $L_{\ell-1}$ even though the size exceeds T. It only creates L_ℓ when the size of $L_{\ell-1}$ exceeds $10T$. It then still keeps a size T of SSTables in $L_{\ell-1}$ and moves at least $9T$ of SSTables to L_ℓ. In this case, ELECT ensures that the last level L_ℓ always contains a sufficiently large number of SSTables (almost 90% of all SSTables across all levels in our case) and maintains the effectiveness of redundancy transitioning. Note that if the LSM-tree grows and adds a new level, the current erasure-coded SSTables in the previous last level will be moved to the new last level.

4.1.2 Parity Node Selection

ELECT applies cross-SSTable encoding on k uncoded SSTables (called *data SSTables*) from k nodes (called *data nodes*) and generates $n-k$ coded SSTables (called *parity SSTables*) that are stored in $n-k$ nodes (called *parity nodes*). Before encoding, ELECT first selects the set of parity nodes to which the parity SSTables are distributed. The selection process should satisfy the following requirements: (i) for fault tolerance, the parity nodes should be distinct from the data nodes; (ii) for load balancing, the parity SSTables are evenly distributed across all nodes after parity node selection; and (iii) for scalability, the parity nodes can be deterministically selected by individual nodes without centralized coordination.

To satisfy the above requirements, ELECT forms each coding group over n consecutive nodes in the hash ring, say $N_{i \bmod M}, N_{(i+1) \bmod M}, \cdots, N_{(i+n-1) \bmod M}$ for $0 \leq i < M$, where M is the total number of nodes, the first k nodes are the data nodes, and the following $n-k$ nodes are the parity nodes. Also, each node locally maintains a monotonic sequence number Q (initialized as zero). Specifically, for each SSTable in the primary LSM-tree that is selected by N_i ($0 \leq i < M$) for erasure coding, N_i selects a *leader parity node* N_p, which will be responsible for computing and sending the parity SSTables (§4.1.3) to $n-k-1$ other parity nodes. It computes p as:

$$p = (i + (Q \bmod k) + 1) \bmod M, \qquad (1)$$

and increments the sequence number Q by one for each SSTable being selected for erasure coding. Note that the selection of SSTables is based on their priorities (§4.2).

We explain via an example the idea behind Equation (1). From Figure 1 (where $M=6$), we use $(6,4)$ RS coding. Thus, N_0 (deterministically) selects a leader parity node from N_1, N_2, N_3, and N_4 in a round-robin fashion for encoding its SSTables as Q increases; similarly, N_1 selects a leader parity node from N_2, N_3, N_4, and N_5, and so forth. Inversely, each node N_i ($0 \leq i < M$) in the whole system will be selected as a leader parity node by k nodes $N_{(i-1) \bmod M}, N_{(i-2) \bmod M}, \cdots, N_{(i-k) \bmod M}$, which now become the k data nodes of a coding group; for example, in Figure 3, N_4 serves as a leader parity node for N_0, N_1, N_2, and N_3 in a coding group under $(6,4)$ RS coding. Since a large-scale storage system typically contains multiple coding groups, ELECT ensures that all coding groups are distributed across different sequences of n consecutive nodes. Finally, each leader parity node (say N_p) will be the first parity node of a coding group, and the remaining $n-k-1$ parity nodes of the coding group are the $n-k-1$ succeeding nodes of N_p along the clockwise direction of the hash ring.

4.1.3 Cross-SSTable Encoding

Encoding workflow. Each of the k data nodes of a coding group sends an SSTable to the leader parity node, which is determined by the sequence number Q according to Equation (1) (§4.1.2). Upon receiving the k data SSTables, the

leader parity node encodes them into $n-k$ parity SSTables, stores one of the parity SSTables locally, and sends the remaining parity SSTables to the other $n-k-1$ parity nodes. The parity nodes store the parity SSTables as separate files outside of their LSM-trees. For example, from Figure 3, consider a coding group for $(6,4)$ RS coding with $k=4$ data nodes N_0, N_1, N_2, and N_3, all of which share the same leader parity node N_4. N_0 sends an SSTable (say S) to N_4. Then, N_4 encodes S together with other SSTables (from N_1, N_2, and N_3, respectively) to generate the parity SSTables (say P and P'). N_4 stores P locally and sends P' to another parity node N_5.

The leader parity node also generates a metadata structure, called `ECMeta`, for the coding group to support failure reconstruction (§5). The `ECMeta` contains n 4-tuples, each of which describes a data/parity SSTable in the coding group, including: (i) the cryptographic hash (e.g., SHA-256) of the content of the data component of the SSTable (32 bytes), (ii) the SSTable size (4 bytes), (iii) the identifier of the node that stores the SSTable (4 bytes), and (iv) the position in the coding group (indexed from 0 to $n-1$) (4 bytes). In particular, we borrow the idea from deduplication [47] and use the SSTable hash as the unique identifier to search for the SSTable during reconstruction, assuming that the hash collisions for distinct SSTables are practically unlikely. The leader parity node then sends the `ECMeta` of each data SSTable to the corresponding R nodes in the replication group. Upon receiving the `ECMeta`, each of the R nodes includes the `ECMeta` in the metadata component of the corresponding SSTable.

Compaction-triggered parity updates. When a primary LSM-tree undergoes compaction, its data SSTables (in the last level) may be updated. ELECT needs to update the parity SSTables of the same coding group to maintain consistency.

Consider a primary LSM-tree that undergoes compaction. Let $S_0, S_1, \cdots, S_u$ be the old data SSTables before compaction, and $S'_0, S'_1, \cdots, S'_v$ be the new data SSTables after compaction, where u and v are the numbers of old data SSTables and new data SSTables, respectively. Without loss of generality, the two sequences of data SSTables $(S_0, S_1, \cdots, S_u)$ and $(S'_0, S'_1, \cdots, S'_v)$ are ordered by (non-overlapping) key ranges. ELECT pairs each of the old and new data SSTables as (S_0, S'_0), (S_1, S'_1), and so forth. If $u < v$ (i.e., there exist more new data SSTables), the extra new data SSTables are simply added as regular SSTables to the primary LSM-tree without erasure coding; if $u > v$ (i.e., there exist fewer new data SSTables due to deleted KV pairs), ELECT pairs each extra old data SSTables with zero-filled dummy SSTables. The dummy SSTables can later be replaced by the new data SSTables that correspond to the same leader parity node.

For each pair of old and new SSTables, ELECT reads the `ECMeta` of the old data SSTable to identify the corresponding leader parity node. It sends the pair to the leader parity node, which updates the parity SSTables of the same coding group based on delta-based parity updates (similar to read-modify-writes in RAID) [10] and sends out the updated `ECMeta`.

4.1.4 Secondary Replica Removal

ELECT removes the secondary replicas from the secondary LSM-trees after cross-SSTable encoding to reclaim storage space. Since the LSM-trees of different nodes perform compaction asynchronously, they may have distinct SSTables. It is important to avoid incorrectly removing KV pairs from the secondary replicas, especially for the KV pairs that are updated after cross-SSTable encoding.

After cross-SSTable encoding, for each primary LSM-tree, ELECT generates a *key list* for each data SSTable, where the key list includes the keys in the SSTables and the corresponding written timestamps; note that the timestamps are already provided by Cassandra to identify the latest versions of KV pairs among replicas. It sends the key list to the other secondary LSM-trees that contain the secondary replicas of the data SSTable. For each secondary LSM-tree, ELECT finds all SSTables in the last level whose KV pairs are covered by the key list. It removes only the KV pairs that are either the current or older versions with respect to the timestamps specified in the key list. Note that ELECT creates new SSTables and tracks the key ranges of the removed KV pairs in the metadata components of the new SSTables, so as to be compatible with the LSM-tree management under replication. Finally, it reconstructs the SSTables for the remaining KV pairs. If the current versions of all KV pairs indicated by the key list are removed, the key list is also removed.

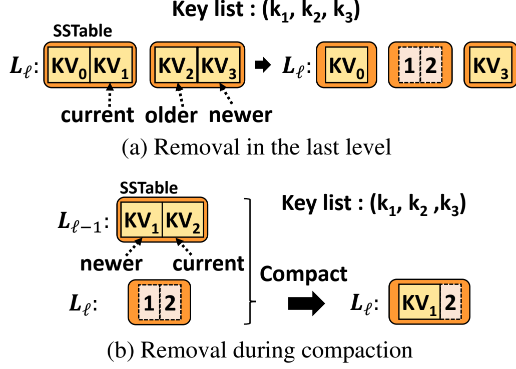
Figure 4(a) shows the replica removal workflow in the last level of a secondary LSM-tree. Suppose that the secondary LSM-tree has four KV pairs in the last level L_ℓ, denoted by KV_0, KV_1, KV_2, and KV_3, whose keys are k_0, k_1, k_2, and k_3, respectively, while the four KV pairs are stored in two SSTables (KV_0, KV_1) and (KV_2, KV_3). Now, suppose that the secondary LSM-tree receives a key list (k_1, k_2, k_3), whose timestamps indicate that KV_1 is the current version (i.e., same timestamp), KV_2 is older, and KV_3 is newer. Thus, ELECT removes KV_1 and KV_2. It also creates an SSTable that tracks the key ranges for the deleted KV_1 and KV_2.

Note that the secondary LSM-tree may not remove the current version of a KV pair (e.g., KV_2) as indicated in the key list, as the KV pair to be removed is not yet moved to the last level due to asynchronous compaction of different nodes. Figure 4(b) shows a case where the second last level $L_{\ell-1}$ has a newer version KV_1 and the current version KV_2 with respect to the timestamps in the key list. During compaction, ELECT removes KV_2 and adds KV_1 to the last level L_ℓ.

4.2 Hotness Awareness

ELECT incorporates hotness awareness into redundancy transitioning and data offloading.

Hotness-aware redundancy transitioning. ELECT monitors the hotness of each SSTable based on two metrics: (i) the access frequency, which refers to the number of reads issued to the SSTable as measured by Cassandra, and (ii) the lifetime, which refers to the elapsed time since the SSTable



(a) Removal in the last level

(b) Removal during compaction

Figure 4: Secondary replica removal in ELECT.

creation. Among the SSTables in the last level of the primary LSM-tree in each node, an SSTable is said to have a higher priority to be selected for encoding (§4.1.3) if it has a lower access frequency and (if a tie exists) a longer lifetime. ELECT selects the SSTables with higher priorities for encoding based on a storage saving target (§4.3).

Cold-data offloading. ELECT dynamically offloads SSTables from the hot tier to the cold tier based on the hotness of SSTables, so as to further mitigate the storage overhead in the hot tier. First, it offloads parity SSTables with long lifetimes as they only affect parity updates (§4.1.3) and failure reconstruction. Second, after all parity SSTables are offloaded, it selectively offloads the data SSTables with higher priorities. The exact numbers of data and parity SSTables being offloaded depend on a storage saving target (§4.3). Note that if an SSTable is selected to be offloaded, only the SSTable's data component is moved, while its metadata component remains in the hot tier to serve read and compaction operations. Furthermore, when a read or compaction operation touches an SSTable in the cold tier, ELECT retrieves the SSTable's data component from the cold tier to the hot tier.

4.3 Balancing Storage-Performance Trade-Off

Redundancy transitioning and data offloading alleviate the storage overhead in the hot tier, yet they also incur performance overhead compared with replicating all data in the hot tier. To balance the trade-off between the storage savings and access performance, ELECT introduces a configurable *storage saving target* α with respect to when all SSTables are replicated, so as to control the number of SSTables involved in redundancy transitioning and data offloading. Specifically, α is a fractional value between zero and one, such that a larger α implies that more SSTables are erasure-coded and offloaded from the hot tier to the cold tier, and vice versa.

Quantifying storage overhead. We approximate the storage overhead in the hot tier based on the number of SSTables in a single primary LSM-tree in a node. Let C_{all} be the total number of SSTables in the primary LSM-tree, C_{last} be the number of SSTables in the last level of the LSM-tree, C_{rt} be the number of data SSTables in the last level being converted from replication to erasure coding, and C_{pm} and C_{dm} be the numbers of parity SSTables and data SSTables being offloaded to the cold tier, respectively.

We now quantify the actual storage size of a replication group (in terms of the number of SSTables), assuming that the storage load is balanced (i.e., all nodes have the same number of SSTables in their respective primary LSM-trees). ELECT replicates $C_{all} - C_{rt}$ SSTables and encodes C_{rt} SSTables, so their storage usage is $(C_{all} - C_{rt}) \cdot R + C_{rt} \cdot \frac{n}{k}$. It also offloads $C_{pm} + C_{dm}$ SSTables to the cold tier. Thus, the actual storage size of a replication group is:

$$(C_{all} - C_{rt}) \cdot R + C_{rt} \cdot \tfrac{n}{k} - C_{pm} - C_{dm}. \quad (2)$$

When all SSTables are replicated, the actual storage size of a replication group is $C_{all} \cdot R$. To achieve the storage saving target α, our goal is to configure C_{rt}, C_{pm}, and C_{dm}, so that

$$1 - \tfrac{1}{C_{all} \cdot R}[(C_{all} - C_{rt}) \cdot R + C_{rt} \cdot \tfrac{n}{k} - C_{pm} - C_{dm}] \geq \alpha. \quad (3)$$

In ELECT, each node computes C_{rt}, C_{pm}, and C_{dm} independently (without centralized coordination) given C_{all}, C_{last}, and α. Assuming balanced storage loads, the respective values of C_{all} and C_{last} across nodes have very small differences.

Balancing trade-off. ELECT starts with redundancy transitioning to keep all SSTables (replicated or erasure-coded) in the hot tier. If α is not met, it offloads parity SSTables to the cold tier, so that all SSTables remain accessible from the hot tier when no failure occurs; in case of a node failure, parity SSTables are needed for recovery. If α is still not met, ELECT offloads data SSTables to the cold tier. Note that ELECT only offloads erasure-coded SSTables to the cold tier, so α may not be achievable if it is too large.

- *Case 1 (Redundancy transitioning):* ELECT sets $C_{pm} = C_{dm} = 0$ and chooses the largest possible C_{rt} to maximize storage savings. Note that $C_{rt} \leq C_{last}$. From Equation (3), C_{rt} is given by:
$$C_{rt} = \min\{\tfrac{R \cdot C_{all} \cdot \alpha}{R - n/k}, C_{last}\}. \quad (4)$$
- *Case 2 (Offloading of parity SSTables):* ELECT proceeds to Case 2 if α is not met, i.e., $C_{rt} = C_{last}$. It sets $C_{dm} = 0$ and chooses the largest possible C_{pm}. Note that the number of parity SSTables is at most $\frac{n-k}{k} \cdot C_{last}$. From Equation (3), C_{pm} is given by:
$$C_{pm} = \min\{R \cdot C_{all} \cdot \alpha - (R - \tfrac{n}{k}) \cdot C_{last}, \tfrac{n-k}{k} \cdot C_{last}\}. \quad (5)$$
- *Case 3 (Offloading of data SSTables):* ELECT proceeds to Case 3 if α is still not met, i.e., $C_{pm} = \frac{n-k}{k} \cdot C_{last}$. It chooses the largest possible C_{dm}. Note that $C_{dm} \leq C_{last}$. From Equation (3), C_{dm} is given by:
$$C_{dm} = \min\{C_{all} \cdot R \cdot \alpha - (R-1) \cdot C_{last}, C_{last}\}. \quad (6)$$

5 Implementation

We implement ELECT in Java based on Cassandra v4.1.0 [3], with around 27 K lines of code of modifications to Cassandra's codebase (which consists of 1.25 M lines of code), by adding redundancy transitioning, hotness monitoring, data offloading, full-node recovery, and degraded reads/writes.

We implement the erasure coding operations based on Intel's Intelligent Storage Acceleration Library [27] and link the operations with Cassandra through the Java Native Interface.

Consistent reads/writes. Under replication, Cassandra supports consistent reads/writes based on the configurable read/write consistency levels, which specify the number of nodes in a replication group that need to acknowledge a read/write request. ELECT maintains the same read/write workflows for replicated KV pairs as in Cassandra. For writes, ELECT performs the same consistent writes as in Cassandra since it always writes KV pairs via replication. For reads, after receiving enough acknowledgments according to the read consistency level, if a KV pair is replicated, ELECT follows the same consistent read path as in Cassandra; if a KV pair is erasure-coded, ELECT always returns the KV pair from the primary LSM-tree or issues degraded reads (see below) if the KV pair is unavailable. ELECT currently does not verify reads for erasure-coded KV pairs.

Full-node recovery. Suppose that a node crashes and all its LSM-trees are lost. ELECT performs recovery in a new node on a per-LSM-tree basis. To recover a primary LSM-tree, ELECT retrieves the secondary LSM-tree from another alive node to the new node. The SSTables from the lowest to the second last level in the LSM-tree are replicated and can be directly recovered from their replicas. For the SSTables in the last level being erasure-coded, ELECT retrieves k data or parity SSTables of the same coding group based on the `ECMeta` from the other alive nodes or the cloud to decode the lost SSTables. To recover a secondary LSM-tree, ELECT retrieves a primary LSM-tree or a secondary LSM-tree from the other nodes in the same replication group; if a primary LSM-tree is retrieved, ELECT removes the data components of SSTables that are erasure-coded, as the secondary LSM-tree only keeps their metadata components (§4.1.4).

Degraded reads. Suppose that ELECT receives a degraded read to an unavailable KV pair. ELECT relays the read request to another alive node in the same replication group, with a flag indicating the KV pair is unavailable. If the KV pair is stored with replication, the alive node directly returns the KV pair; otherwise, if the KV pair is stored with erasure coding, the alive node decodes the SSTable containing the KV pair by retrieving k data or parity SSTables of the same coding group from the other alive nodes according to the `ECMeta`.

Degraded writes. Suppose that ELECT receives a degraded write to a failed node. It follows Cassandra to apply the *hinted handoff* mechanism [4], which allows the replay of a write to a failed node that returns online.

Limitations. ELECT does not support incremental recovery for individual SSTables as in Cassandra. Under replication, Cassandra builds a Merkle tree [41] in each node to detect inconsistencies among replicas for any failed SSTable recovery. Since ELECT includes erasure-coded SSTables in LSM-trees, it needs a revised Merkle tree that addresses both replication and erasure coding.

ELECT also does not currently support dynamic topology changes. We consider a possible approach for supporting topology changes in ELECT as follows. For replicated KV pairs, ELECT can relocate replicas when the nodes join or leave as in Cassandra. For erasure-coded KV pairs, ELECT can relocate some of the erasure-coded SSTables to keep them in consecutive nodes in the hash ring (§4.1.2). As in consistent hashing, ELECT should only relocate the KV pairs stored in the adjacent nodes of each joining/leaving node in the hash ring instead of all SSTables of the whole storage system, so as to mitigate the relocation overhead.

6 Evaluation

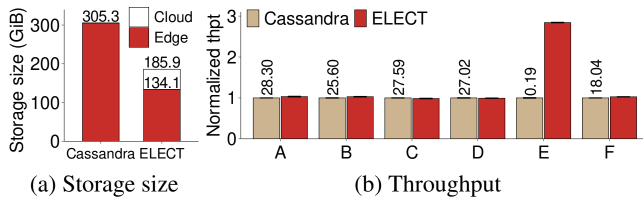
We show via evaluation that ELECT reduces the storage overhead of Cassandra and maintains high performance.

6.1 Methodology

Testbed. We consider an *edge-cloud setting*, where the edge serves as the hot tier and the cloud serves as the cold tier. Specifically, we conduct evaluation on Alibaba Cloud [38]. We set up $M = 10$ edge nodes and multiple (up to 32) client nodes in the same geographical region. Each node is deployed on an `ecs.i3g.2xlarge` instance with eight 2.5 GHz vCPUs, 32 GiB RAM, 447 GiB SSD, and Ubuntu 22.04 LTS. All nodes are connected with a 3 Gbps network, with a network latency of no more than 1 ms. We also deploy the cloud on the Alibaba Object Storage Service in a different geographical region. Our measurement shows that the network latency between the two regions is at least 45 ms.

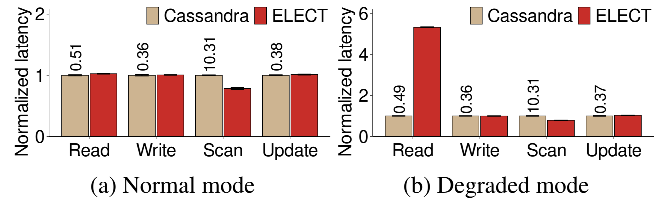
Default settings. We compare ELECT with the vanilla Cassandra. We configure Cassandra with triple replication ($R = 3$) and store all replicas in the edge nodes. We also configure ELECT with triple replication and the $(n,k) = (6,4)$ RS code, and enable all features under a storage saving target $\alpha = 0.6$. For both systems, we fix the SSTable size as 4 MiB [28,64]. We disable SSTable compression for accurate storage size calculation. We set the read and write consistency levels as one and three, respectively (i.e., each of the reads and writes needs to be acknowledged by one node and all three replica nodes, respectively) for strong consistency. All other parameters remain the same as the defaults in Cassandra.

We use the benchmarking tool YCSB [16] to generate different types of workloads. By default, in Exp#1, we load 100 M 1-KiB KV pairs with 24-byte keys and 1000-byte values into storage before each experiment and generate 10 M requests; in the subsequent experiments that focus on examining the performance of individual KV operations, we load 10 M 1-KiB KV pairs and generate 1 M requests, while the performance trends remain stable as in Exp#1 even we use smaller workloads. In all experiments, the requests by default follow a Zipf distribution with a Zipfian constant of 0.99 (default in YCSB). Also, we run YCSB clients in two client nodes, each of which has eight YCSB client threads to simulate concurrent requests from multiple clients.



(a) Storage size (b) Throughput

Figure 5: Exp#1: YCSB core workloads. Throughput results are normalized by Cassandra's throughput (above each bar) in KOPS.



(a) Normal mode (b) Degraded mode

Figure 6: Exp#2: Benchmarking of KV operations. Results are normalized by Cassandra's latencies (above each bar) in milliseconds.

Our experiments consider normal (without failures) and degraded (with failures) modes. For degraded mode, we crash two edge nodes via the `kill -9` command.

We plot the average results over five runs, with error bars showing the 95% confidence intervals under the Student's t-distribution (note that some error bars may be invisible due to small deviations).

6.2 Overall Analysis

Exp#1 (YCSB core workloads). We first compare the overall storage overhead and performance of Cassandra and ELECT using the six YCSB core workloads [16], namely A (50% reads, 50% writes), B (95% reads, 5% writes), C (100% reads), D (95% reads, 5% writes), E (95% scans, 5% writes), and F (50% reads, 50% read-modify-writes). Each workload (except D) follows a Zipf distribution, while Workload D reads the latest written KV pairs. For ELECT, we measure both edge-only and overall edge-cloud storage sizes.

Figure 5 shows the storage size and throughput results. ELECT achieves 56.1% storage savings (in the edge only) and 39.1% overall storage savings (in both the edge and cloud) compared with Cassandra. The actual edge storage savings of ELECT are slightly less than $\alpha = 0.6$, as it also maintains metadata components for deleted KV pairs in the secondary LSM-trees after redundancy transitioning. The metadata components of LSM-trees in ELECT account for 6.9% of its edge storage size (not shown in the figure); note that no metadata components are offloaded to the cloud (§4.2).

In terms of performance, both Cassandra and ELECT have similar throughput (with up to 3% differences) in all workloads except E. For Workload E, which is scan-intensive, ELECT achieves a 2.84× throughput gain over Cassandra. The reason is that ELECT reduces individual LSM-tree sizes and hence I/O amplification through decoupled replication management, so the number of SSTables being accessed is also reduced [57,68]. Such reduced access overhead has more prominent performance improvements to scans, which read a range of KV pairs.

Exp#2 (Benchmarking of KV operations). We evaluate the average latencies of specific KV operations, including reads, writes, scans, and updates. We load 10 M 1-KiB KV pairs and issue 1 M requests for each type of KV operations (§6.1). We consider both normal and degraded modes. For degraded mode, we measure the performance of all requests (including normal and degraded requests) when the system is in degraded mode with edge node failures. For example, if a read encounters a non-failed node, it is a normal read; if it encounters a failed node, it becomes a degraded read and ELECT recovers the SSTable that contains the requested key.

Figure 6 shows the results. Note that Cassandra keeps almost identical performance in both normal and degraded modes as it keeps all replicated storage in the edge. In normal mode (Figure 6(a)), ELECT maintains similar performance as in Cassandra in reads, writes, and updates with up to 2.7% of higher average latencies; it reduces the average scan latency of Cassandra by 21.5% (see Exp#1). In degraded mode (Figure 6(b)), ELECT still has similar performance of Cassandra in writes and updates with up to 3.3% higher average latencies and reduces the average scan latency of Cassandra by 21.1%. However, ELECT incurs a latency increase of 5.32× in reads over Cassandra, mainly due to the retrieval of SSTables from the cloud to the edge for recovery if the degraded reads are issued to the KV pairs in the last LSM-tree level.

We further examine the read and scan results in degraded mode. For reads, we observe that both Cassandra and ELECT have very similar 99th-percentile latencies at about 1.7 ms (not shown in the figure), meaning that most reads can be served in the edge and have small latency differences. Some degraded reads need to retrieve SSTables from the cloud, and such requests increase the average read latency in degraded mode. Unlike reads, ELECT still shows performance gains in scans (which include normal and degraded reads to a range of KV pairs) as in normal mode. The reason is that most unavailable SSTables are recovered in the early stage of scans, so the overall adverse impact on scans is much mitigated as opposed to reads.

6.3 System-level Analysis

Exp#3 (Performance breakdown). We break down the performance of writes, reads in normal mode, and reads in degraded mode. Each write comprises (i) writing to the WAL, (ii) writing to the MemTable, (iii) flushing the MemTable, (iv) compaction, (v) redundancy transitioning, and (vi) data offloading. Each read in normal or degraded mode comprises (i) reading from the MemTable, (ii) reading from the block cache, (iii) reading from SSTables, and (iv) recovery (for degraded reads). We measure the time of each step across all nodes and obtain the average results on processing 1 MiB of writes/reads based on the workloads as described in §6.1. Since the steps are performed in parallel, the actual time spent

Steps	Cassandra	ELECT
Write		
WAL	21.32 ± 0.76 ms	21.84 ± 0.28 ms
MemTable	37.98 ± 1.73 ms	40.84 ± 0.13 ms
Flushing	16.95 ± 0.29 ms	17.70 ± 0.18 ms
Compaction	205.87± 2.21 ms	169.03 ± 3.23 ms
Transitioning	-	239.05 ± 2.69ms
Offloading	-	162.84 ± 12.05 ms
Read in normal mode		
Cache	17.05 ± 0.27 ms	18.35 ± 0.34 ms
MemTable	20.78 ± 0.95 ms	23.20 ± 0.61 ms
SSTables	182.69 ± 2.53 ms	177.55 ± 0.60 ms
Read in degraded mode		
Cache	17.41 ± 0.33 ms	18.75 ± 0.18 ms
MemTable	21.54 ± 0.66 ms	23.38 ± 0.46 ms
SSTables	184.39 ± 1.67 ms	184.14 ± 2.35 ms
Recovery	-	1957.64 ± 34.16 ms

Table 1: Exp#3: Performance breakdown. We show the average latency of each step for processing 1 MiB of writes/reads and the corresponding 95% confidence interval.

on an operation is less than the sum of times of all steps.

Table 1 shows the performance breakdown. Most common steps in Cassandra and ELECT have similar latencies, except for compaction in writes, ELECT has a smaller latency by 17.9%. The reason is that ELECT decouples replication management into multiple LSM-trees and reduces the I/O amplifications in compaction [57, 68]. The redundancy transitioning has the highest average latency among all steps, while the offloading has a low average latency since only a fraction of data is transmitted from the edge to the cloud. However, both redundancy transitioning and data offloading are performed in the background and incur limited overhead to writes, so ELECT has similar write performance as Cassandra (see Exp#1 and Exp#2).

For reads in normal mode, both Cassandra and ELECT have similar performance in each step. For reads in degraded mode, ELECT is bottlenecked by the recovery step (with 1957.64 ms per MiB).

Exp#4 (Full-node recovery). We evaluate full-node recovery in Cassandra and ELECT. We crash one edge node, delete all its data, start a new edge node, and use the `nodetool` command to recover all lost data into the new edge node. We evaluate the recovery performance of different loaded data sizes, including 10 GiB, 20 GiB, and 30 GiB (i.e., 10 M, 20 M, and 30 M 1-KiB KV pairs). For fair comparisons, we disable the Merkle tree operation in Cassandra (which is not supported in ELECT (§5)). Figure 7 shows the full-node recovery times. The recovery times of Cassandra and ELECT increase almost linearly. ELECT incurs about 50% higher recovery time than Cassandra since it needs to retrieve data and parity SSTables from other edge nodes or the cloud to decode the lost SSTables in the primary LSM-tree.

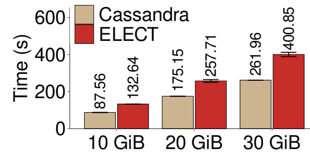


Figure 7: Exp#4: Full-node recovery time.

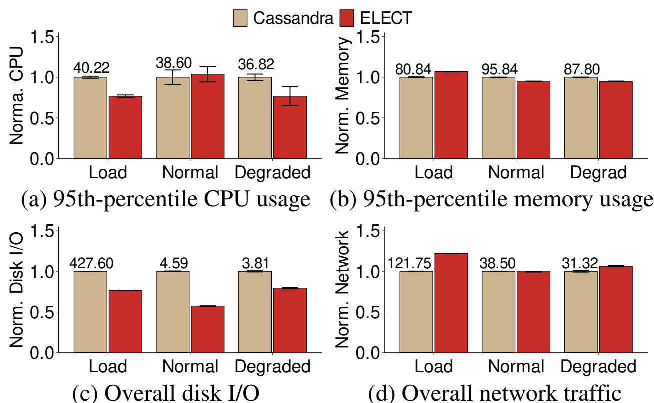
Steps	Time
Copy	13.54± 0.22 s
Retrieve	373.98± 11.61 s
Decode	13.34 ± 0.48 s

Table 2: Exp#4: Recovery time breakdown for 30 GiB data.

We further provide a breakdown of the full-node recovery time of ELECT into three steps: (i) copying replicated SSTables from other edge nodes; (ii) retrieving data and parity SSTables for decoding; and (iii) decoding the lost SSTables. Table 2 shows the full-node recovery time breakdown of ELECT for 30 GiB data. The recovery performance is network-bound, in which retrieving the data and parity SSTables occupies 93.3% of the total recovery time.

Exp#5 (Resource usage). We compare the CPU usage, memory usage, disk I/O size, and network traffic of Cassandra and ELECT. We consider three settings: (i) loading KV pairs until the SSTables reach a stable state, (ii) running in normal mode without failures, and (iii) running in degraded mode with two node failures. After loading KV pairs, we issue 1 M requests, including reads, writes, updates, and scans, with one-fourth of all requests each. We measure the CPU usage, memory usage, disk I/O, and network usage in all alive edge nodes through the Linux system tools, namely `top`, `free`, `iostat`, and `iftop`, respectively. We collect the resource usage data every 1 s. We show the 95th-percentile CPU usage and memory usage as well as the overall disk I/O size and network traffic size.

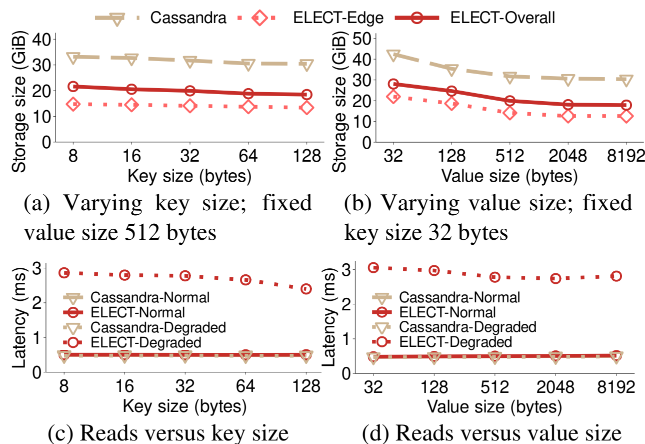
Figure 8 shows the results. Figure 8(a) shows that ELECT has 23.4% and 23.3% less 95th-percentile CPU usage than Cassandra in load and degraded operations, respectively. ELECT mainly performs network transmissions in redundancy transitioning and data offloading, both of which involve less CPU usage (in each 1-second interval). However, ELECT has long durations in both redundancy transitioning and data offloading (Table 1), so its total CPU time is still higher than Cassandra's. Figure 8(b) shows that ELECT slightly increases the 95th-percentile memory usage by 7.0% during the load operation, since it needs extra memory space for erasure coding. It also slightly reduces the memory usage in normal and degraded modes by 4.9% and 5.1%, respectively, as it reduces the LSM-tree size through decoupled replication management. Figure 8(c) shows that ELECT reduces the disk I/O size of Cassandra by 23.6% in the load operation, as it reduces the LSM-tree size and hence I/O amplifications through decoupled replication management, and also reduces the compaction overhead of the secondary LSM-trees by writing fewer KV pairs to the last LSM-tree level. Note that ELECT still has higher disk I/O size in degraded mode than in normal mode due to the reconstruction overhead in erasure coding. Figure 8(d) shows that ELECT has similar network traffic to Cassandra in normal mode, while



(a) 95th-percentile CPU usage (b) 95th-percentile memory usage

(c) Overall disk I/O (d) Overall network traffic

Figure 8: Exp#5: Resource usage. All results are normalized by Cassandra's actual resource usage results (numbered atop the bars), including CPU usage (%), memory usage (GiB), disk I/O size (GiB), and network traffic (GiB).



(a) Varying key size; fixed value size 512 bytes (b) Varying value size; fixed key size 32 bytes

(c) Reads versus key size (d) Reads versus value size

Figure 9: Exp#6: Impact of key and value sizes.

it incurs 22.1% and 6.3% higher network traffic than Cassandra in load and degraded operations, respectively. In the load operation, ELECT distributes SSTables for redundancy transitioning and offloads SSTables to the cloud; in degraded mode, it retrieves SSTables to recover unavailable SSTables.

6.4 Parameter Sensitivity Analysis

Exp#6 (Impact of key and value sizes). We evaluate the impact of different key and value sizes on Cassandra and ELECT to show that ELECT still maintains storage savings for different key/value sizes. Note that we fix the total size of KV pairs loaded to storage as 10 GiB, so the total number of KV pairs decreases as the key size or value size increases.

Figures 9(a) and 9(b) show the actual storage sizes of Cassandra and ELECT for various key sizes with a fixed value size 512 bytes and various value sizes with a fixed key size 32 bytes, respectively. The actual storage sizes of both systems decrease as the key and value sizes increase. The edge and overall storage savings of ELECT over Cassandra increase from 55.5% to 56.0% and from 34.9% to 39.3%, respectively, as the key size increases from 8 bytes to 128 bytes, and from 48.2% to 58.5% and from 33.9% to 41.1%, respectively, as the value size increases from 32 bytes to 8 KiB. The reason is that larger key and value sizes reduce the amount of metadata for SSTable maintenance. This reduces the storage overhead after redundancy transitioning in ELECT.

Figures 9(c) and 9(d) show the average read latencies in normal and degraded modes for various key and value sizes. The read latencies of Cassandra and ELECT in normal mode and the read latency of Cassandra in degraded mode are similar as the KV pairs can be directly accessed. However, ELECT has much higher average read latencies in degraded mode than Cassandra, especially for small key and value sizes (e.g., 6.4× when the key and value sizes are both 32 bytes), since smaller key and value sizes increase the number of KV pairs stored in an SSTable and hence the query overhead.

Exp#7 (Impact of storage saving target). We evaluate the impact of target storage saving α on ELECT, by varying α from 0.1 to 0.9. Our results demonstrate the trade-off between storage savings and access performance.

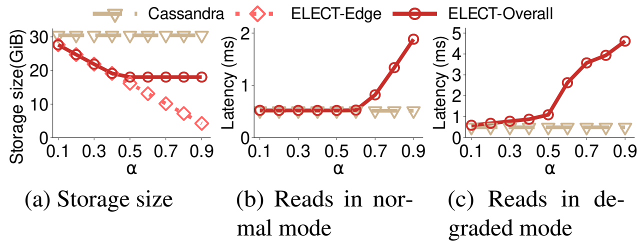
Figure 10(a) shows the edge-only and overall storage sizes of ELECT. As α increases, the edge storage savings over Cassandra increase from 9.2% to 86.0%, and differ from α by no more than 4%. ELECT offloads SSTables from the edge to the cloud when $\alpha \geq 0.5$, yet the overall storage size (in both the edge and cloud) of ELECT remains unchanged when $\alpha \geq 0.5$ and its savings over Cassandra stay at 40.8%. The reason is that redundancy transitioning only applies to the SSTables in the last LSM-tree level and cannot further reduce the storage sizes of replicated SSTables in the lower LSM-tree levels.

Figure 10(b) shows the average read latencies in normal mode. The read latency of ELECT remains stable as α increases from 0.1 to 0.6, but increases significantly from 0.53 ms to 1.89 ms when α increases from 0.6 to 0.9. The reason is that after offloading data SSTables to the cloud (Case 3 in §4.3), reads to the primary LSM-tree retrieve data SSTables back from the cloud and are slowed down by edge-cloud communication. We pose the parameter sensitivity analysis for 99th-percentile latencies as future work.

Figure 10(c) shows the average read latencies in degraded mode. As α increases from 0.1 to 0.5, the average latency of ELECT increases from 0.59 ms to 1.09 ms, as more SSTables are involved in redundancy transitioning and degraded reads trigger more decoding operations. As α further increases, ELECT starts to offload data SSTables to the cloud, and the degraded reads are further slowed down due to the retrieval of data SSTables from the cloud for decoding. When $\alpha = 0.9$, the average latency increases to 4.62 ms.

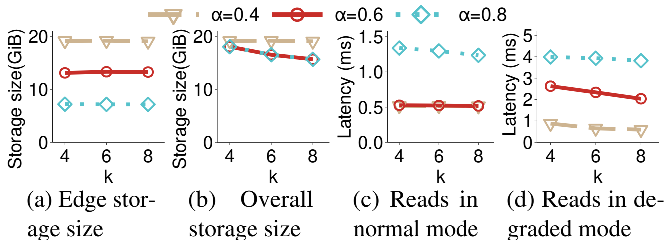
Exp#8 (Impact of coding parameters). We study the impact of coding parameters on ELECT by varying k and fixing $n = k+2$. We also consider different values of α. We focus on the edge and overall storage sizes as well as the average read latencies in normal and degraded modes.

Figure 11 shows the results. For a fixed α, even with



(a) Storage size (b) Reads in normal mode (c) Reads in degraded mode

Figure 10: Exp#7: Impact of storage saving target. Note that the results for Cassandra remain unaffected by α but are included in the plots for comparisons.



(a) Edge storage size (b) Overall storage size (c) Reads in normal mode (d) Reads in degraded mode

Figure 11: Exp#8: Impact of coding parameters.

different values of k, ELECT still maintains similar edge storage sizes with no more than 1.7% differences (Figure 11(a)). Although a larger k (with the fixed $n-k$) implies smaller redundancy, the storage saving target α also determines the actual edge storage size. Thus, the edge storage size remains almost unaffected by different values of k for a fixed α.

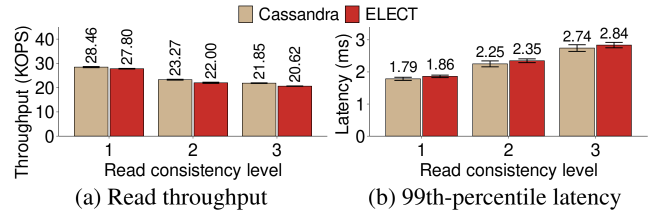
The overall storage size (Figure 11(b)) of ELECT drops from 18.0 GiB to 15.7 GiB when k increases from 4 to 8, for both $\alpha = 0.6$ and $\alpha = 0.8$, as a larger k generates fewer parity SSTables and reduces the overall storage size for a large α; note that the overall storage size remains unaffected for different values of k for $\alpha = 0.4$.

For a fixed α, the reads latencies in normal and degraded modes are also similar for different values of k, albeit slight latency decreases in degraded mode as k increases (Figures 11(c) and 11(d)). Intuitively, a larger k implies higher reconstruction overhead, as k SSTables are retrieved for reconstruction in RS codes (§2.3). On the other hand, a larger k also implies that fewer parity SSTables are generated and offloaded to the cloud, and hence a degraded read retrieves fewer parity SSTables from the cloud. This leads to slightly improved read performance in degraded mode for a large k.

This experiment aims to show the applicability of ELECT for different values of k. A more detailed analysis on the trade-off between α and k is our future work.

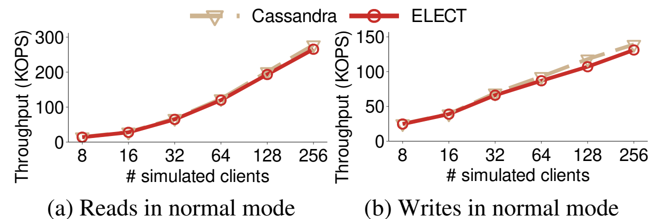
Exp#9 (Impact of read consistency level). We show how ELECT preserves consistent reads in Cassandra (§5). We vary the read consistency level from one to three, while the write consistency level remains three (under triple replication). We focus on the throughput and 99th-percentile latency of normal reads; for the latter, we show the impact of waiting for responses from multiple replicas.

Figure 12 shows the results versus the read consistency level for both Cassandra and ELECT. As the read consistency level increases from one to three (i.e., each read needs to wait for the responses from more replicas), the average read throughput of both systems (Figure 12(a)) decreases by 23.2% for Cassandra and 25.8% for ELECT, while the 99th-percentile read latencies for both systems (Figure 12(b)) increase by around 50%. The results suggest that ELECT maintains similar read performance as in Cassandra under consistent reads.



(a) Read throughput (b) 99th-percentile latency

Figure 12: Exp#9: Impact of read consistency level.



(a) Reads in normal mode (b) Writes in normal mode

Figure 13: Exp#10: Impact of number of clients.

Exp#10 (Impact of number of clients). We examine the read/write performance of ELECT in normal mode as the number of clients increases. We vary the number of client nodes (deployed in different instances) from 1 to 32, while each client node runs eight client threads, so the maximum number of simulated clients reaches 256. Each simulated client issues 100 K KV requests.

Figure 13 shows the throughput versus the number of simulated clients for both Cassandra and ELECT. Both systems show similar increasing throughput trends as the number of simulated clients increases. ELECT has slightly less throughput than Cassandra, by 4% in normal reads and 5.7% in writes when the number of simulated clients is 256, due to the redundancy transitioning overhead.

6.5 Discussion

We discuss the performance of ELECT in other aspects that are currently not explicitly evaluated.

Varying skewness in workloads. We currently focus on skewed workloads as observed in practice [6, 9, 16, 62]. With less skewed workloads, reads access larger portions of the key space. Thus, more reads are issued to the last LSM-tree level, and ELECT may see performance drops in degraded mode as it applies erasure coding to the last LSM-tree level. Note that ELECT does not directly determine the hotness of KV pairs by monitoring their access patterns, while the read and write patterns may have different distributions [6, 62]. Thus, the actual performance of ELECT may be greatly affected by the real-world access patterns.

Varying LSM-tree sizes. ELECT supports different LSM-tree sizes as it still encodes SSTables in the last LSM-tree level. It is expected to achieve higher storage savings for larger LSM-trees since the number of SSTables increases exponentially across LSM-tree levels and the last LSM-tree level contains more SSTables.

Impact on reliability. The increase in the recovery time of ELECT (Exp#4) degrades reliability (e.g., in terms of mean-time-to-data-loss (MTTDL)). On the other hand, since ELECT offloads some erasure-coded KV pairs to the cold tier, which is in general more reliable than the hot tier (e.g., when edge nodes serve as the hot tier versus the cloud serves as the cold tier in edge-cloud storage), the reliability can be improved. The actual impact on reliability due to redundancy transitioning remains an open issue.

Impact of LSM-tree compression. Our evaluation disables compression to fairly measure ELECT's storage savings. ELECT still works with compression enabled and is expected to achieve storage savings. Since Cassandra performs compression on SSTables, ELECT can collect k compressed data SSTables and pad them with zeroes to match the maximum size of the k compressed data SSTables, so as to generate parity SSTables via erasure coding. Note that such padded zeros are only for erasure coding compatibility. They need not be stored in data SSTables and will not add storage overheads.

Comparisons with CassandrEAS [8]. CassandrEAS also extends Cassandra with erasure coding, but does not consider redundancy transitioning. CassandrEAS reportedly has much higher read and write latencies than Cassandra [8]. Our evaluation also finds that CassandrEAS (based on its open-source version) incurs high storage overhead for small values due to extra metadata for erasure coding (e.g., 1.6× storage overhead for 24-byte keys and 64-byte values compared with 3-way replication in Cassandra).

7 Related Work

Replication in distributed KV stores. Replication is commonly used in modern distributed KV stores [5, 18, 34, 46]. Several studies propose new replica management mechanisms to support high-throughput and strongly consistent writes [56], reduce data loss rates [15], improve query performance [22, 54], and reduce I/O amplification in LSM-tree compaction [57, 68]. In particular, ELECT has the similar ideas of DEPART [68] and Tebis [57] to separate the LSM-tree management for replicas, but focuses on synchronizing the views of replicas for efficient redundancy transitioning. Both DEPART and Tebis do not consider erasure coding.

Erasure coding in distributed KV stores. Erasure coding has been extensively used in distributed KV stores. Some studies apply replication for keys and metadata as well as erasure coding for values for persistent [8, 32, 33, 44] and in-memory [13, 37] KV stores. Some approaches [14, 65] apply erasure coding across whole objects (including keys, values, and metadata) for further storage savings, but they are applied for in-memory KV stores and their performance is guaranteed with memory access. Erasure coding is also recently explored for disaggregated memory [35, 69].

In the context of storage tiering, EC-Cache [49] and C2DN [61] also apply erasure coding to caching and content delivery networks, respectively. EC-Cache performs self-encoding on large objects and keeps erasure-coded chunks across cache servers, while C2DN replicates small objects and applies erasure coding to large objects. We emphasize that ELECT differs from EC-Cache and C2DN in both problem formulation and design techniques. Regarding problem formulation, EC-Cache and C2DN aim for load balancing using erasure coding for high performance, while ELECT considers redundancy transitioning (from replication to erasure coding) for storage savings. Regarding design techniques, EC-Cache and C2DN are centralized (EC-Cache manages cache servers with a centralized coordinator, while C2DN uses a cluster-local load balancer), while ELECT performs decentralized parity node selection (§4.1.2). ELECT also addresses selective data offloading (§4.2) and configurable storage savings (§4.3), both of which are not addressed by EC-Cache and C2DN.

Redundancy transitioning. Earlier studies consider the transitioning between replication and erasure coding on fixed-size blocks in RAID [58] and distributed file systems [23, 36], while ELECT considers variable-size KV pairs. Furthermore, ELECT builds on Cassandra, a decentralized KV store, while the above studies [23, 36, 58] are centralized. Some studies consider the transitioning between erasure codes with different coding parameters to trade between performance and redundancy overhead [55, 59, 60, 63] or between reliability and redundancy overhead [29, 30]. Convertible codes [39] are new erasure codes that minimize the transitioning I/O. ELECT applies redundancy transitioning from replication to erasure coding to balance the storage-performance trade-off.

8 Conclusions

We design ELECT, a distributed KV store that enables erasure coding tiering, to make a case for storage-efficient, high-performance, and fault-tolerant KV storage. ELECT adopts a hybrid redundancy approach by replicating hot KV pairs and erasure-coding cold KV pairs. It also selectively offloads them from the hot tier to the cold tier. It is tunable with a single storage saving target parameter to balance the trade-off between storage savings and access performance. Experiments on Alibaba Cloud demonstrate the storage savings and performance efficiency of ELECT compared with Cassandra.

Acknowledgements. We thank our shepherd, Gala Yadgar, and the anonymous reviewers for their comments. This work was supported in part by National Key R&D Program of China (2022YFB4501200), Key Research and Development Program of Hubei Province (2021BAA189), National Natural Science Foundation of China (61972073, 62302175, and 62332004), and Research Grants Council of Hong Kong (GRF 14214622 and AoE/P-404/18).

References

[1] Amazon. Amazon Elastic Block Store. https://aws.amazon.com/ebs/.

[2] Amazon. Amazon Simple Storage Service. https://aws.amazon.com/s3/.

[3] Apache. Cassandra 4.1.0. https://github.com/apache/cassandra/releases/tag/cassandra-4.1.0.

[4] Apache. Cassandra documentation - hints. https://cassandra.apache.org/doc/4.1/cassandra/operating/hints.html.

[5] Apache. HBase. https://hbase.apache.org/.

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proc. of ACM SIGMETRICS*, 2012.

[7] Burton Howard Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 12(7):422--426, 1970.

[8] Viveck R. Cadambe, Kishori M. Konwar, Muriel Medard, Haochen Pan, Lewis Tseng, and Yingjian Wu. CassandrEAS: Highly available and storage-efficient distributed key-value store with erasure coding. In *Proc. of NCA*, 2020.

[9] Zhichao Cao and Siying Dong. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proc. of USENIX FAST*, 2020.

[10] Jeremy C. W. Chan, Qian Ding, Patrick P. C. Lee, and Helen H. W. Chan. Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage. In *Proc. of USENIX FAST*, 2014.

[11] 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. In *Proc. of USENIX OSDI*, 2006.

[12] Batyr Charyyev, Engin Arslan, and Mehmet Hadi Gunes. Latency comparison of cloud datacenters and edge servers. In *Proc. of IEEE GLOBECOM*, 2020.

[13] Haibo Chen, Heng Zhang, Mingkai Dong, Zhaoguo Wang, Yubin Xia, Haibing Guan, and Binyu Zang. Efficient and available in-memory KV-store with hybrid erasure coding and replication. *ACM Trans. on Storage*, 13(3):25, 2017.

[14] Liangfeng Cheng, Yuchong Hu, and Patrick P. C. Lee. Coupling decentralized key-value stores with erasure coding. In *Proc. of ACM SOCC*, 2019.

[15] Asaf Cidon, Stephen Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Proc. of USENIX ATC*, 2013.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of ACM SOCC*, 2010.

[17] Couchbase. Couchbase automated data partitioning. https://www.couchbase.com/blog/what-exactly-membase/.

[18] 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 *Proc. of ACM SOSP*, 2007.

[19] Alexandros G. Dimakis, P. Brighten Godfrey, Yunnan Wu, Martin J. Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE Trans. on Information Theory*, 56(9):4539--4551, 2010.

[20] Partha Dutta, Rachid Guerraoui, and Ron R. Levy. Optimistic erasure-coded distributed storage. In *Proc. of DISC*, 2008.

[21] EdgeKV. Augment image and video manager with edge compute. https://techdocs.akamai.com/edgekv/docs/augment-image-and-video-manager-with-edge-compute.

[22] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A distributed, searchable key-value store. In *Proc. of ACM SIGCOMM*, 2012.

[23] Bin Fan, Wittawat Tantisiriroj, Lin Xiao, and Garth A. Gibson. Diskreduce: Replication as a prelude to erasure coding in data-intensive scalable computing. *Technical Report, CMU-PDL-11-112, Carnegie Mellon University Parallel Data Laboratory*, 2011.

[24] Daniel Ford, François Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *Proc. of USENIX OSDI*, 2010.

[25] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proc. of ACM SOSP*, 2003.

[26] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *Proc. of USENIX ATC*, 2012.

[27] Intel. Intelligent storage acceleration library. https://github.com/intel/isa-l.

[28] Jeeyoon Jung and Dongkun Shin. Lifetime-leveling LSM-tree compaction for ZNS SSD. In *Proc. of ACM HotStorage*, 2022.

[29] Saurabh Kadekodi, Francisco Maturana, Sanjith Athlur, Arif Merchant, K. V. Rashmi, and Gregory R. Ganger. Tiger: Disk-adaptive redundancy without placement restrictions. In *Proc. of USENIX OSDI*, 2022.

[30] Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Juncheng Yang, K. V. Rashmi, and Gregory R. Ganger. PACEMAKER: Avoiding HeART attacks in storage clusters with disk-adaptive redundancy. In *Proc. of USENIX OSDI*, 2020.

[31] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of ACM STOC*, 1997.

[32] Kishori M. Konwar, N. Prakash, Erez Kantor, Nancy Lynch, Muriel Médard, and Alexander A. Schwarzmann. Storage-optimized data-atomic algorithms for handling erasures and errors in distributed storage systems. In *Proc. of IEEE IPDPS*, 2016.

[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 *Proc. of IEEE MSST*, 2015.

[34] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review*, 44(2):35--40, 2010.

[35] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Hydra: Resilient and highly available remote memory. In *Proc. of USENIX FAST*, 2022.

[36] Runhui Li, Yuchong Hu, and Patrick P. C. Lee. Enabling efficient and reliable transition from replication to erasure coding for clustered file systems. *IEEE Trans. on parallel and distributed systems*, 28(9):2500--2513, 2017.

[37] Witold Litwin, Rim Moussa, and Thomas Schwarz. LH*RS---a highly-available scalable distributed data structure. *ACM Trans. on Database System*, 30(3):769--811, 2005.

[38] Alibaba Cloud Computing Co. Ltd. Alibaba cloud. `https://www.alibabacloud.com/`.

[39] Francisco Maturana and K. V. Rashmi. Convertible codes: Enabling efficient conversion of coded data in distributed storage. *IEEE Trans. on Information Theory*, 68(7):4392 -- 4407, 2022.

[40] Ruben Mayer, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. FogStore: Toward a distributed data store for fog computing. In *Proc. of IEEE FWC*, 2017.

[41] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Proc. of CRYPTO*, 1987.

[42] Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal De Lara. CloudPath: A multi-tier cloud computing framework. In *Proc. of ACM/IEEE SEC*, 2017.

[43] 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 *Proc. of USENIX OSDI*, 2014.

[44] Nicolas Nicolaou, Viveck Cadambe, N. Prakash, Andria Trigeorgi, Kishori Konwar, Muriel Medard, and Nancy Lynch. ARES: Adaptive, reconfigurable, erasure coded, atomic storage. In *Proc. of IEEE ICDCS*, 2019.

[45] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33:351--385, 1996.

[46] PingCAP. TiKV. `https://tikv.org`.

[47] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *Proc. USENIX FAST*, 2002.

[48] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proc. of ACM SOSP*, 2017.

[49] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proc. of USENIX OSDI*, 2016.

[50] 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.

[51] David Reinsel. How you contribute to today's growing datasphere and its enterprise impact. `https://blogs.idc.com/2019/11/04/how-you-contribute-to-todays-growing-datasphere-and-its-enterprise-impact/`, Nov 2019.

[52] Mahadev Satyanarayanan. The emergence of edge computing. *IEEE Computer*, 50(1):30--39, 2017.

[53] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637 -- 646, 2016.

[54] Amy Tai, Michael Wei, Michael J. Freedman, Ittai Abraham, and Dahlia Malkhi. Replex: A scalable, highly available multi-index data store. In *Proc. of USENIX ATC*, 2016.

[55] Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. Fast and strongly-consistent per-item resilience in key-value stores. In *Proc. of ACM EuroSys*, 2018.

[56] Robbert Van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. of USENIX OSDI*, 2004.

[57] Michalis Vardoulakis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tebis: index shipping for efficient replication in lsm key-value stores. In *Proc. of EuroSys*, 2022.

[58] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Trans. on Computer Systems*, 14(1):108--136, 1996.

[59] Si Wu, Zhirong Shen, and Patrick P. C. Lee. Enabling I/O-efficient redundancy transitioning in erasure-coded KV stores via elastic Reed-Solomon codes. In *Proc. of IEEE SRDS*, 2020.

[60] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A. Pease. A tale of two erasure codes in HDFS. In *Proc. of USENIX FAST*, 2015.

[61] Juncheng Yang, Anirudh Sabnis, Daniel S. Berger, K. V. Rashmi, and Ramesh K. Sitaraman. C2DN: How to harness erasure codes at the edge for efficient content delivery. In *Proc. of USENIX NSDI*, 2022.

[62] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proc. of USENIX OSDI*, 2020.

[63] Qiaori Yao, Yuchong Hu, Liangfeng Cheng, Patrick P. C. Lee, Dan Feng, Weichun Wang, and Wei Chen. StripeMerge: Efficient wide-stripe generation for large-scale erasure-coded storage. In *Proc. of IEEE ICDCS*, 2021.

[64] 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 *Proc. of USENIX FAST*, 2019.

[65] Matt M. T. Yiu, Helen H. W. Chan, and Patrick P. C. Lee. Erasure coding for small objects in in-memory KV storage. In *Proc. of ACM SYSTOR*, 2017.

[66] Heng Zhang, Mingkai Dong, and Haibo Chen. Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *Proc. of USENIX FAST*, 2016.

[67] Heng Zhang, Shaoyuan Huang, Mengwei Xu, Deke Guo, Xiaofei Wang, Victor C. M. Leung, and Wenyu Wang. How far have edge clouds gone? a spatial-temporal analysis of edge network latency in the wild. In *Proc. of IEEE/ACM IWQoS*, 2023.

[68] Qiang Zhang, Yongkun Li, Patrick P. C. Lee, Yinlong Xu, and Si Wu. DEPART: Replica decoupling for distributed key-value storage. In *Proc. of USENIX FAST*, 2022.

[69] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-tolerant far memory. In *Proc. of USENIX OSDI*, 2022.

A Artifact Appendix

Abstract

ELECT is a distributed KV store that enables erasure coding tiering based on the LSM-tree. It adopts a hybrid redundancy approach that carefully combines replication and erasure coding with respect to the LSM-tree layout. Its prototype builds on Cassandra.

Scope

Our artifact can be used to validate the concepts and designs of ELECT presented in the paper. It is a research-driven prototype and has several limitations, such as the inability to support dynamic topology changes and incremental recovery, that restrict its direct applicability in production.

Contents

The artifact consists of two sub-directories:

- `src/`, which includes both the implementation of the ELECT prototype and a simple object storage backend, such that ELECT can be connected with the object storage backend in a local cluster; and
- `scripts/`, which includes both the evaluation scripts and the YCSB benchmarking tool, such that the key and value sizes are configurable.

The artifact also contains a README file that specifies the prerequisites for the testbed and dependencies, steps for building and configuring the ELECT prototype and YCSB benchmark tool, and detailed instructions for artifact evaluation.

Hosting

The artifact is accessible from GitHub at `https://github.com/adslabcuhk/elect`. The version we provided for the artifact evaluation is marked with the `v1.0` tag.

Requirements

Hardware dependencies

To successfully run the end-to-end experiments with our prototype and evaluation scripts, a minimum of eight machines are recommended. These machines need to be connected via a network, such that they are reachable from each other. For each machine, we recommend quad-cores, 16 GiB of memory and above, and an SSD. We need at least six machines that form the distributed KV store ELECT and use the default erasure coding parameters (n,k)=$(6,4)$. In addition, we have one machine that acts as a server node for storing cold data in the cold tier, and one machine for running the YCSB benchmark tool.

Software dependencies

Our artifact is developed and tested on Ubuntu 22.04 LTS with the following software dependencies:

- The ELECT prototype and YCSB benchmark tool: `openjdk-11-jdk`, `openjdk-11-jre`, `ant`, `ant-optional`, `maven`.
- Erasure coding: `clang`, `llvm`, `libisal-dev`.
- Evaluation scripts: `ansible`, `bc`, `python3`, `python3-pip`, `cassandra-driver`, `numpy`, `scipy`.

Testbed Setup

Please follow the steps below:

- Download the artifact from the URL: `https://github.com/adslabcuhk/elect/releases`.
- Extract the files using `tar -zxvf elect-1.0.tar.gz` and navigate into the package directory with `cd`.
- Modify the `scripts/settings.sh` file according to the `AE_INSTRUCTION.md`.
- Set up the machines with the provided scripts via `bash scripts/setup.sh full` (the setup takes about 40 minutes, depending on the hardware configurations).

For the detailed setup, configuration instructions, and troubleshooting, please refer to the `README.md` in the artifact repository. The `README.md` file provides comprehensive instructions on the manual setup process and the solutions to some common issues.

Evaluation

Artifact Claims

The performance results may vary from those in our paper due to different factors, such as cluster sizes, machine specifications, operating systems, and software packages. However, we expect that ELECT still demonstrates comparable performance to Cassandra in regular operations, while significantly reducing hot-tier storage overhead.

Experiments

To reproduce the results presented in the paper, please refer to the `AE_INSTRUCTION.md` file and follow the instructions provided in the `Evaluation` section.

Exp#1 (YCSB core workloads). *Expected outcome:* Exp#1 produces the results as shown in Figure 5, which illustrates that ELECT achieves similar performance as Cassandra in YCSB core workloads while significantly reducing the hot-tier storage overhead. In addition, ELECT outperforms Cassandra in workload E, which consists of 95% scan operations, due to replication decoupling. *Approximate runtime:* 20 compute hours.

Exp#2 (Benchmarking of KV operations). *Expected outcome:* Exp#2 produces the results as shown in Figure 6, which illustrates that ELECT achieves similar performance as Cassandra in common KV operations. Similar to Exp#1, ELECT still outperforms Cassandra in the scan operations due to replication decoupling. *Approximate runtime:* 5 compute hours.

Exp#3 (Performance breakdown). *Expected outcome:* Exp#3 produces the results as shown in Table 1, which illustrates that ELECT has similar latencies in most common steps as in Cassandra. *Approximate runtime:* 5 compute hours.

Exp#4 (Full-node recovery). *Expected outcome:* Exp#4 produces the results as shown in Figure 7 and Table 2, which illustrates that ELECT incurs medium recovery overhead due to the need for retrieving data and parity SSTables from other nodes or the cold tier. *Approximate runtime:* 14 compute hours.

Exp#5 (Resource usage). *Expected outcome:* Exp#5 produces the results as shown in Figure 8, which illustrates that ELECT only increases the memory and network usage when loading data due to redundancy transitioning and cold-data offloading. In addition, for CPU usage, the 95%-percentile CPU utilization will be less than Cassandra since the redundancy transitioning and cold-data offloading consist of a large amount of network transmission with long duration. *Approximate runtime:* 5 compute hours.

Exp#6 (Impact of key and value sizes). *Expected outcome:* Exp#6 produces the results as shown in Figure 9, which illustrates that ELECT still maintains storage savings for different key/value sizes. *Approximate runtime:* 40 compute hours.

Exp#7 (Impact of storage saving target). *Expected outcome:* Exp#7 produces the results as shown in Figure 10, which illustrates that ELECT can balance the storage overhead and performance according to the storage saving target. *Approximate runtime:* 45 compute hours.

Exp#8 (Impact of coding parameters). *Expected outcome:* Exp#8 produces the results as shown in Figure 11, which illustrates that ELECT can adapt to different erasure coding parameters. *Approximate runtime:* 12 compute hours.

Exp#9 (Impact of read consistency level). *Expected outcome:* Exp#9 produces the results as shown in Figure 12, which illustrates that ELECT supports consistent reads. *Approximate runtime:* 5 compute hours.

Exp#10 (Impact of number of clients). *Expected outcome:* Exp#10 produces the results as shown in Figure 13, which illustrates that ELECT supports multi-client KV operations. *Approximate runtime:* 5 compute hours.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

MinFlow: High-performance and Cost-efficient Data Passing for I/O-intensive Stateful Serverless Analytics

Tao Li[1], Yongkun Li[1*], Wenzhe Zhu[1], Yinlong Xu[1,3], John C. S. Lui[2]

[1]*University of Science and Technology of China* [2]*The Chinese University of Hong Kong*
[3]*Anhui Province Key Laboratory of High Performance Computing, USTC*

Abstract

Serverless computing has revolutionized application deployment, obviating traditional infrastructure management and dynamically allocating resources on demand. A significant use case is I/O-intensive applications like data analytics, which widely employ the pivotal "shuffle" operation. Unfortunately, the shuffle operation poses severe challenges due to the massive PUT/GET requests to remote storage, especially in high-parallelism scenarios, leading to high performance degradation and storage cost. Existing designs optimize the data passing performance from multiple aspects, while they operate in an isolated way, thus still introducing unforeseen performance bottlenecks and bypassing untapped optimization opportunities. In this paper, we develop MinFlow, a holistic data passing framework for I/O-intensive serverless analytics jobs. MinFlow first rapidly generates numerous feasible multi-level data passing topologies with much fewer PUT/GET operations, then it leverages an interleaved partitioning strategy to divide the topology DAG into small-size bipartite sub-graphs to optimize function scheduling, further reducing over half of the transmitted data to remote storage. Moreover, MinFlow also develops a precise model to determine the optimal configuration, thus minimizing data passing time under practical function deployments. We implement a prototype of MinFlow, and extensive experiments show that MinFlow significantly outperforms state-of-the-art systems, FaaSFlow and Lambada, in both the job completion time and storage cost.

1 Introduction

Serverless computing, or simply "serverless", represents a transformative cloud-computing model that dramatically streamlines application deployment. Within this paradigm, the burdensome tasks of traditional infrastructure management recede into the background as cloud providers dynamically allocate resources, billing solely for the consumed computing

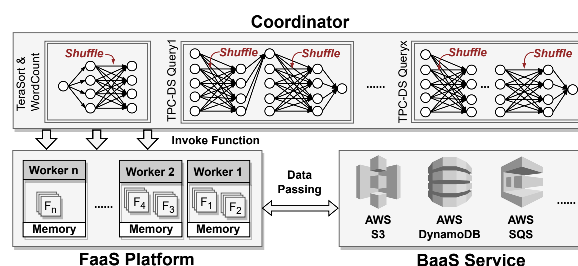


Figure 1: Serverless Computing Framework. *Circles represent sub-tasks, and each link between circles represents a data transfer, consisting of one PUT plus one GET.*

power. Platforms such as AWS Lambda [5] and Azure Functions [31] exemplify this shift, facilitating the seamless execution of code in response to specific triggers. As we navigate the evolving expanse of cloud technologies, the prominence of serverless is undeniable, marking a significant change in the development, deployment, and scaling of modern applications. This shift becomes particularly noteworthy when considering I/O-intensive applications like data analytics [21, 22, 35]. Embedded in this analytical landscape are frameworks like Google's MapReduce [15] and Apache Spark [44].

In data analytics, the "*shuffle*" operation is pivotal for data passing between stages. Notably, over half of Facebook's daily analytics entails at least one shuffle operation [45]. Given the stateless nature of serverless, data are largely passed through remote Object Stores like S3 [8], during which each pair of sender and receiver functions involve a PUT and a GET operation. However, shuffle's all-to-all connectivity, i.e., each function should pass its output to all functions in the next stage, usually leads to a huge number of PUT/GET requests, especially under high parallelism of functions. For instance, with 500 functions, one can anticipate 500×500=250,000 PUTs and an equal number of GETs, a total of 500,000 requests. Due to the request rate caps of S3, excessive PUT/GET operations risk exceeding these limits, causing prolonged delays. For example, in the Pocket framework, shuffle can dominate, taking up 62% of the time for certain jobs [22]. Worse yet,

[*]Yongkun Li is the corresponding author.

while S3's storage capacity is affordable, the cost tied to massive PUT/GET operations can escalate very high.

In data analytics, optimizing the shuffle operation has led to a myriad of solutions, each has its unique trade-off. Though these solutions propose diverse optimization strategies, they lack a comprehensive and integrative consideration, resulting in suboptimal performance. First, the approach of using private storage has been utilized [22, 35], wherein shuffle is conducted through self-maintained storage such as ElastiCache clusters [4]. While it offers enhanced shuffle speed by granting users exclusive ownership of the storage medium, the ensuing costs are considerably elevated. Additionally, the onus of intricate cluster management falls back on the developers, somewhat undermining the convenience of serverless computing. The method of leveraging intra-worker memory offers another alternative [13, 25], harnessing over-provisioned local memory in workers for faster shuffle operations. However, its applicability remains tethered to functions situated within the same worker, and due to the all-to-all data passing requirement between functions, only a small portion of data passing can be performed via the local memory of workers. Lastly, the technique of utilizing multi-level shuffle [32, 34], inspired by the mesh networks from HPC (High Performance Computing) [23], endeavors to streamline shuffle operations. Yet, it incurs multiplied data to be transmitted, making the bandwidth limit on the function side a new bottleneck, especially when the size of the data input is large. In conclusion, rather than offering a holistic solution, existing techniques operate in isolated realms, sometimes incurring unintended costs or introducing unforeseen bottlenecks. Moreover, the absence of a systematic exploration implies that potential optimizations still remain untapped.

In this paper, we propose MinFlow, a unified data passing framework for I/O-intensive analytics jobs atop serverless, which pinpoints globally optimal configuration to simultaneously achieve high performance and low cost. MinFlow contains the following key innovations:

- It optimizes the data passing topology by first segmenting functions into adaptive groups and then progressively converging the groups to get integrated multi-level topologies. This methodology not only greatly reduces the number of PUT/GET operations, but also provides the flexibility of selecting from a broader range of feasible topologies under real-world settings.
- It develops an interleaved partitioning strategy to optimize the function scheduling. Specifically, it partitions a multi-level topology into bipartite structures, and schedules functions in units of the bipartite sub-graphs so as to allow the localization of data passing within workers.
- It leverages a precise model to pinpoint the optimal configuration according to real function deployments, i.e., the best combination of topology and function scheduling, so as to simultaneously minimize the number of PUTs/GETs and the storage cost.

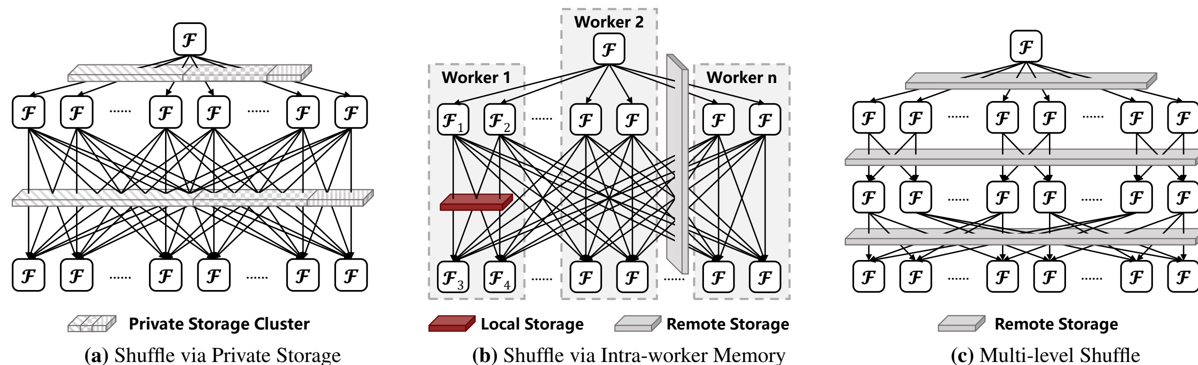
We implement a prototype of MinFlow open-sourced at https://github.com/lt2000/MinFlow and conduct extensive experiments based on Amazon cloud service. Our experiments using the benchmarks of TeraSort, TPC-DS, and WordCount show that MinFlow significantly outperforms state-of-the-art works in both the shuffling performance and storage cost. For example, in high-parallelism case of 600 mapper and 600 reducer functions for 200GB TeraSort, MinFlow reduces the shuffle time by 66.62% and 89.22%, compared to Lambada [32] and FaaSFlow [25], respectively, and it also reduces the storage cost by 86% and 98.71%, respectively.

2 Background and Motivation

2.1 Background

Serverless Computing Framework. As the building blocks of serverless, FaaS and BaaS (e.g., Amazon Lambda [5] and S3 object store [8]) respectively empower users to directly invoke predefined functions in containers and access remote back-end services via RESTful APIs. When employing serverless services, a common practice is first to decouple applications' states and compute logic, then delegate them to BaaS-side storage and FaaS-side functions separately (see Figure 1). Merits of the architecture are twofold. First, the separation of storage and computation and the containerized functions greatly facilitate scaling up/down compute resources as needed (e.g., to tackle bursty workloads). Second, it provides a fine-grained "pay-as-you-go" billing model that charges for actually used resources rather than the reserved amount; e.g., Amazon Lambda provides billing increments of one millisecond during function execution [6]. Due to all its virtues, an increasing number of applications have embraced the architecture, including Web, IoT, data analytics, etc. [7, 18, 31].

Data Analytics atop Serverless. Data analytics aims at efficiently processing huge amounts of data as specified to obtain desired results, and it has been employed in a wide range of domains, including scientific computing, machine learning, large-scale graph computations, etc. [39]. To offer essential scalability and fault-tolerance, mainstream data analytics frameworks [15, 30, 43] commonly adopt the bulk-synchronous-parallel model (BSP) [40], which divides a job into consecutive stages, each stage composed of parallel subtasks. When each stage is completed, the intermediate results are transferred to the next stage, via communication primitives such as *shuffle* and *broadcast* [14, 19] for further computation. Thereby, the workflow of jobs employing BSP can be represented as DAGs, as illustrated in Figure 1. To deploy data analytics jobs atop serverless platforms, users typically first declare the job's workflow to a coordinator using configuration files. Then, the coordinator assumes control, activating functions to perform consecutive stages sequentially, with sub-tasks within each stage executed by parallel functions. Users receive notifications when the whole job is completed.



(a) Shuffle via Private Storage **(b)** Shuffle via Intra-worker Memory **(c)** Multi-level Shuffle

Figure 2: Existing Approaches.

Notably, since serverless functions are unable to directly communicate with each other, data transmission between stages is realized via remote back-end storage, typically S3, rather than direct peer-to-peer (P2P) data passing. Since analytics workloads typically have a wide variance of resource needs over time [34, 35, 46], traditional physical/VMs deployments can frequently suffer from resource wastage or performance degradation. On the contrary, benefiting from the elasticity and fine-grained billing model of FaaS, the job's computation can be easily accelerated by splitting each stage into more sub-tasks assigned to parallel functions, at a significantly lower cost and higher performance. Thereby, a lot of research works have focused on running data analytics based on serverless [13, 20, 22, 25, 27, 32, 34, 35, 45].

2.2 Dilemma Caused by *Shuffle*

In data analytics, *Shuffle* is the most common primitive for passing data between adjacent stages. Prior research shows that more than 50% of daily data analytics jobs at Facebook involve at least one shuffle operation [45]. As shown in Figure 1, during the shuffle process, each sub-task in the previous stage distributes its output to all sub-tasks in the next stage. Such an all-to-all data passing method would greatly "break down" the intermediate results, causing proliferating requests: for instance, if the parallelism of stages is N, then at least $2N^2$ object PUTs/GETs are required to pass the intermediate results since there would be N^2 links between stages and each link represents a PUT plus a GET. This causes problems in two aspects. First, it significantly degrades the performance. Due to S3's request rate limit (3.5k and 5.5k req/s for writes and reads [10]), the quadratic $2N^2$ PUTs/GETs could easily be throttled, especially when N is large. Consequently, while the computation time could be slashed by improving the parallelism N, the entire data analytics process is significantly slowed down by shuffle. For example, in Pocket, over 62% of time is spent shuffling data, while computation only takes 17% of time for 100GB TeraSort [22]. Second, it drastically inflates the cost. Albeit S3 offers cheap storage (0.023 USD$ per GB/month), it incurs large access cost as it charges in increments of single request (0.005/0.0004 USD$ per 1k PUTs/GETs) [9]. As a result, the $2N^2$ PUTs/GETs would rapidly increase the cost as N goes up. In conclusion, both the elasticity and economy of serverless get severely impeded by the shuffle.

2.3 Existing Approaches

Existing approaches bypass or mitigate S3's throttling by (1) *performing Shuffle via a private storage cluster*, (2) *performing Shuffle via intra-worker memory*, or (3) *using multi-level Shuffle to decrease the number of PUTs/GETs*.

Shuffle via Private Storage. As a public cloud storage service shared by numerous users and applications, S3 inherently allocates a limited request rate to each single user, to guarantee fairness and avoid interference among tenants. A straightforward way to eliminate this restriction is to replace S3 with self-maintained private storage, for example, ElastiCache clusters [4]. This provides the user an exclusive ownership of the storage service, thereby greatly improving shuffle speed. However, losing S3's sharing economy and fine-grained billing model often leads to a significant increase in cost. As Pocket [22] suggests, the cost is 100 times higher than S3 for sorting jobs. Therefore, some remedies have been proposed to mitigate the surging cost, e.g., as shown in Figure 2(a), Pocket [22] and Locus [35] dynamically rightsizing resources, and combine high-end and cheap storage media to achieve better trade-offs between performance and cost.

Shuffle via Intra-worker Memory. Another way to bypass S3's throttling is to reclaim and leverage over-provisioned memory in workers to accelerate shuffle [13, 25]. More specifically, data passing between functions located in the same worker is performed via its local memory. Take functions F_2 and F_3 co-located in worker W_1 in Figure 2(b) as an example. Suppose data to be passed from F_i to F_j is denoted as $<F_i, F_j>$. To deliver data to F_3, F_2 first puts $<F_2, F_3>$ into W_1's local memory, then F_3 fetches $<F_2, F_3>$ immediately afterwards and finishes the transmission. This approach performs well in both performance and cost, since the reclaimed over-provisioned local memory not only offers much higher

bandwidth and lower latency, but also does not incur extra overhead. The downside is the limitation in its applicability, as only co-located functions can adopt this approach [25].

Multi-level Shuffle. Borrowing ideas from HPC, which achieves all-to-all connections among processors through the k-dimensional Mesh Network [16], Starling [34] and Lambada [32] project shuffle-involved functions onto a k-dimensional mesh with a side length of $\sqrt[k]{N}$ where N refers to the number of functions, and applies the all-to-all collective primitive to subsets of functions, once for each dimension, to realize all-to-all shuffle among functions. Compared to direct data passing, such a multi-level indirect manner (*ML-Shuffle*) greatly decreases the number of requests, since each request loads a larger volume of data, and only one PUT plus one GET is required for each link. To be more precise, k-level shuffle (*kL-Shuffle*) reduces the number of requests from $2N^2$ to $2kN\sqrt[k]{N}$. For example, in Figure 2(c) we show a *2L-Shuffle* by setting k as 2. As can be seen, only 60 requests are needed, compared to 72 when directly connecting functions (see Figure 2(a)). Due to fewer requests that need to be transmitted through remote S3 during the shuffle, performance degradation caused by S3's throttling gets mitigated, and lower fees are charged as well.

2.4 Limitations

The aforementioned approaches face respective limitations in cost, performance, or applicability. First, to maintain the private storage for faster shuffle, users have to bear additional management works like resource scaling, fault tolerance, etc., which should have been undertaken by serverless, thus violating the easy-of-use principle. Besides, private storage still entails high costs. Although using dedicated NVMe storage seems cost-efficient, the need to mount NVMe devices to VMs [22] and the limited network bandwidth of VMs significantly hinder the speed advantage, requiring the allocation of numerous NVMe instances for performance. To illustrate this, we run the state-of-the-art KV database Apache kvrocks [11] on varying numbers of NVMe instances (EC2 i3.2xlarge, the same as Pocket uses) for shuffle. As depicted in Figure 3, increasing NVMe instances reduces shuffle time but at a significant cost. Pocket also reports that the cost of Pocket-NVMe is 40 times that of S3 for TeraSort [22].

Second, for performing shuffle via intra-worker memory (e.g., FaaSFlow [25]), it's only applicable to functions co-located at the same worker. For analytics jobs, each group of co-located functions represents a sub-graph in the whole workflow DAG, and all groups together make up the whole DAG. As a result, though functions in the same sub-graph can communicate through local memory, due to the all-to-all feature of shuffle, links between sub-graphs still dominate, which necessitates the use of remote storage for data passing. Worse yet, the benefits of memory-assisted data passing could be easily offset by the stragglers caused by slower remote storage. As shown in Figure 3, under different configurations,

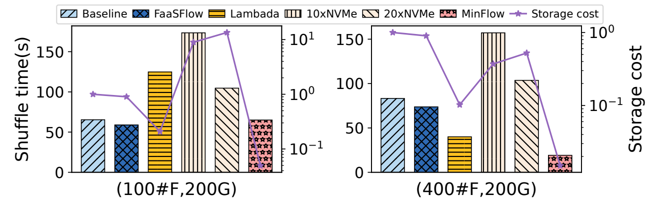


Figure 3: TeraSort Shuffle Time under Different Configurations. *Baseline transfers all intermediate data via S3 and the storage costs are normalized to the baseline.*

the shuffle time of FaaSFlow is only reduced by 9.94% to 11.39% than that of the Baseline.

Regarding *ML-Shuffle* (e.g., Lambada [32]), existing methods based on k-dimensional mesh suffer from an applicability problem, i.e., it mandates a symmetrical Mapper-Reducer setting, which means the number of Mappers and Reducers must be the same (e.g., both are N). Besides, while allowing to adjust the topology with different parameters (e.g., k), they merely set parameters arbitrarily and delegate the tricky task of choosing the optimal parameters to users, which easily leads to sub-optimal performance. For example, while larger k decreases the number of requests more significantly, it brings about multiplied extra data volume to be transferred. Because cloud vendors often assign limited network bandwidth to each function [12, 22, 32], such heavy traffic could exacerbate the problem. On the contrary, smaller k often comes with an unsatisfactory effect on reducing the number of requests. For example, under 100 functions and 200GB input data, the shuffle time of Lambada is 1.91× than that of the Baseline (see Figure 3). Moreover, the 2-level shuffle algorithm can not be easily applied to more levels, and the extension from the 2-level shuffle algorithm to a general k-level one is non-trivial (see §A.3).

Last, as shown in Figure 3, compared to the above three approaches, MinFlow achieves a high-performance and cost-efficient shuffle. We will further carry out extensive experiments to show the benefits of MinFlow in §4.

Inefficiency Analysis. Though a series of optimizing "actions" can be employed, for lack of a systematic understanding, there isn't a judicious "decision maker" that can use them collaboratively. Consequently, multiple factors together decide the efficiency of shuffle, e.g., DAG topology, function scheduling, transmission manner assignment, existing optimizations work in their respective single dimension, paying disproportionate expenses or leaving the rest as a bottleneck. Besides, even for each single dimension, the possible action space is still not fully explored. For instance, current *ML-Shuffle* directly migrates the k-dimensional mesh from the HPC field, whose applicability is strictly limited in the serverless scenario. Last, rather than carefully considering the characteristics of specific analytics jobs and environment variables to select the most appropriate choice, they often merely offer empirical value, e.g., k is set as 2. All these make existing

approaches reach the sub-optimal configuration, leading to degraded performance/cost/ease-of-use.

2.5 Main Idea and Challenges

Main Idea. The key factors deciding analytics jobs' efficiency include function topology represented by the DAG, function scheduling, and the data transmission media. Compared to considering them separately, optimizing them in a unified way greatly helps find the optimal configuration, so as to eradicate bottlenecks from the whole workflow. For instance, *ML-Shuffle* facilitates traffic localization through local memory since the links at each level are more sparse and functions can be co-located more easily to avoid cross-worker data transmission. Also, traffic localization largely absorbs the additional traffic volume induced by *ML-Shuffle*. Therefore, for any given analytics job, our main idea is to first construct the whole configuration space by considering all three dimensions, then derive the optimal configuration from the space, based on user requirements, the task's characteristics, and the serverless platform's rate limit and billing rules.

Challenges. To realize the above idea, we mainly face the following challenges.

- **Constructing *ML-Shuffle* topology space.** To ensure applicability, we must be able to construct the complete topological space for any analytics job, including those with an asymmetric setting of Mappers and Reducers, despite the conventional mesh-based method supposes $\#mapper = \#reducer = N$ and only provides a concrete algorithm for 2-level shuffle[†] . Plus, for a specified analytics job, the complete *ML-Shuffle* topology space contains a number of possible combinations. Thus we need to efficiently construct the *ML-Shuffle* topology space with low overhead.
- **Function co-location and data transmission.** For each possible topology in the space, we need to carefully assign functions to workers to maximize the proportion of leveraging local memory for data passing, while simultaneously ensuring load-balance among workers and avoiding stragglers. This process is equivalent to that of searching for a partitioning scheme that divides the whole DAG into sub-graphs consisting of co-located functions in accordance with requirements, which is an NP-hard problem and especially time-consuming when the number of functions is large.
- **Finding the optimal configuration.** To select the optimal configuration from the space, we need to precisely model the mapping from each configuration to its performance and cost. To achieve this, we must take multiple key factors into consideration, e.g., the analytics job's intermediate data volume and the number of I/O requests, functions' network bandwidth, and remote storage's request rates, some of which can only be obtained at runtime, or be dependent on specific platforms.

[†] *#mapper* and *#reducer* are the number of mappers and reducers, and N is a positive integer.

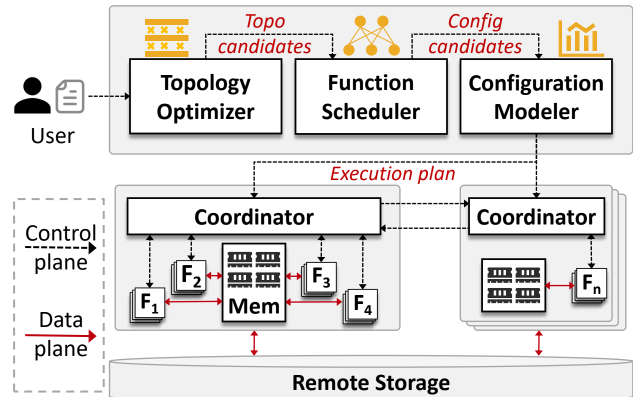


Figure 4: Overview of MinFlow Architecture.

3 MinFlow Design

To optimize the data passing between functions, we propose MinFlow, a unified data passing framework for analytics jobs atop serverless platforms, which seeks the global optimal configuration to achieve high performance and low cost simultaneously. We first introduce the overall architecture (§3.1) and elaborate on each technique in detail (§3.2-§3.4).

3.1 Overview

As Figure 4 illustrates, MinFlow resides in the cloud-side control plane, generating appropriate configuration for specific analytics jobs upon receiving user-submitted workflow specifications which delineate data passing paths between functions and the communication operators executed by these functions. Then, the coordinators deploy and run the task accordingly, upon FaaS and BaaS platforms. Specifically, MinFlow consists of three key components that work collaboratively to meet this goal while tackling the aforementioned challenges at the same time. A brief introduction of the components and their interaction is as follows.

- **Topology Optimizer.** For a given analytics job, it generates equivalent multi-level topologies based on the original single-level topology, via a novel progressively converging method to sidestep the inherent applicability downside of the mesh-based approach. More specifically, all candidates for the ultimate optimal topology, i.e., those with the fewest edges for each possible level, are rapidly constructed by a dynamic programming algorithm, while others are ignored.
- **Function Scheduler.** For each generated candidate topology, the Function Scheduler decides which functions should be co-located at the same worker and passes data through local memory, by dividing the complete topology into sub-graphs. The partitioning must simultaneously achieve load balance, cross-worker traffic minimization, and straggler

avoidance. To solve the NP-hard problem, MinFlow employs a heuristic algorithm to find the near optimal solution quickly.

- **Configuration Modeler.** Configuration Modeler selects the optimal configuration, i.e., that with the shortest estimated completion time and lowest cost for data passing, among candidates. At its core is a mathematical model that factors in key variables, including serverless platform features and analytics job characteristics, to achieve high estimation accuracy. In particular, for those variables needed to be obtained at runtime, it determines them by an efficient and lightweight sampling method.

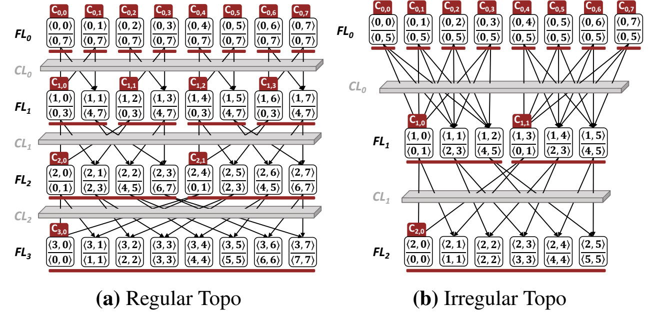
Note that though our current design follows FaaSFlow's distributed function coordination, which employs multiple coordinators to prevent function scheduling from becoming the bottleneck (see Figure 4), MinFlow also applies to the more conventional architecture with a centralized coordinator.

3.2 Topology Optimizer

Progressively Converging ML-Shuffle. Following the mesh-based *ML-Shuffle*, the progressively converging *ML-Shuffle* attempts to generate optimized topology, which is equivalent to the original single-level shuffle directly linking all pairs of mappers and reducers, by adding intermediate functions to reduce the number of required links. In contrast, it aims to offer essential flexibility to search in the complete feasible space for the optimal topology, instead of only providing a single sub-optimal topology as the mesh-based method does [32,34]. Moreover, it's a general k-level shuffle algorithm that allows an asymmetric number of mappers and reducers.

The key idea behind the progressively converging *ML-Shuffle* is a "divide and conquer" strategy. To avoid ambiguity, we clarify that a *kL-Shuffle* network consists of $k+1$ function levels (denoted as *flevel*) and k communication levels (denoted as *clevel*), and Figure 5(a) shows a *3L-Shuffle* network involving four *flevels* and three *clevels*. Rather than projecting functions to a rigid k-dimensional grid, progressively converging first divides functions in the first *flevel* into groups of the same size (initially one) and gradually lets them converge into larger groups in the next *flevel*, while preserving the full connection between each group and its upstream mappers, until all functions in the last *flevel* exist in the same group, thus ultimately achieving global all-to-all connection. For example, as Figure 5(a) illustrates, to build a three-level shuffle when $\#mapper = \#reducer = 8$, functions are respectively divided into 8, 4, 2, and 1 group for *flevel* 0, 1, 2, 3. Suppose we let $C_{i,j}/F_{i,j}$ denote the j-th group/function at *flevel* i, the data in $C_{0,0}$ in turn passes into $C_{1,0}$, $C_{2,0}$, and $C_{3,0}$. Analogously, the data in $C_{0,7}$ passes into $C_{1,3}$, $C_{2,1}$, and $C_{3,0}$. The rest are similar.

More generally, to derive an L-level topology comprising $L+1$ function levels, with each *flevel* having N functions, we divide the functions of the i-th *flevel* into g_i groups, where



(a) Regular Topo **(b)** Irregular Topo

Figure 5: Progressively Converging ML-Shuffle Topology. *Squares represent functions, and the upper and lower tuples inside functions respectively represent the function's id and the data's range.*

$g_i(\,0 \le i \le L)$ meets the following conditions:

$$g_0 = N,\ g_L = 1,\ g_i = d_i \times g_{i+1} \text{ where } d_i \in \mathbb{N}^+ \backslash \{1\}. \quad (1)$$

When converging groups, to preserve the full connection between the new group in the *flevel* $i+1$ and its upstream mappers, for each function in the new group, a unique path between it and any upstream group in the *flevel* i must be guaranteed. To achieve this, we set the receiver functions of $F_{i,j}$ as R:

$$R = \{F_{i+1,k} | \lfloor k/s_{i+1} \rfloor = \lfloor j/s_{i+1} \rfloor \wedge \lfloor k\%s_{i+1}/d_i \rfloor = j\%s_i\}, \quad \text{where } s_i = \lfloor N/g_i \rfloor,\ s_{i+1} = \lfloor N/g_{i+1} \rfloor,\ d_i = \lfloor g_i/g_{i+1} \rfloor. \quad (2)$$

For example, as shown in Figure 5(a), when converging $C_{1,0}$ and $C_{1,1}$ into $C_{2,0}$, we link $F_{1,0}$ to $\{F_{2,0}, F_{2,1}\}$, $F_{1,1}$ to $\{F_{2,2}, F_{2,3}\}$, $F_{1,2}$ to $\{F_{2,0}, F_{2,1}\}$, and $F_{1,3}$ to $\{F_{2,2}, F_{2,3}\}$, thus preserving the full connection between $\{F_{2,0}, F_{2,1}, F_{2,2}, F_{2,3}\}$ with their upstream mappers $\{F_{0,0}, F_{0,1}, F_{0,2}, F_{0,3}\}$. As we see, the link distribution is also kept even to balance the transmission load among functions. Moreover, to ensure the correctness of data passing, each function must carefully partition and distribute received data to the next *flevel* functions. For function $F_{i,j}$, it first shards its data into $|R|$ continuous and equal-sized parts, then orderly assigns them to the receiver functions, i.e., functions in its R. For instance, as illustrated in Figure 5(a), $F_{0,0}$ shards its data $\langle 0,7 \rangle$ into two parts $\langle 0,3 \rangle$ and $\langle 4,7 \rangle$, and passes $\langle 0,3 \rangle$ and $\langle 4,7 \rangle$ to $F_{1,0}$ and $F_{1,1}$, respectively.

Notably, the flexibility of the progressively converging method lies in the setting of $D = \{d_i | 0 \le i \le L-1\}$, since it determines $G = \{g_i | 0 \le i \le L\}$ and any G that satisfies condition Equation (1) corresponds to a unique valid multi-level topology. In other words, by adjusting D we can easily derive a space containing multiple optional topologies, which may vary in edge distribution and the number of *clevels* and thus have different preferences for the number of requests, data transmission volume, etc. For example, the data passing volume is obviously proportional to L since each additional *clevel* incurs one more intermediate data transmission, and combining Equation (2) we have that the number of edges is $N \times \sum_{i=0}^{L-1} d_i$, by doubling which we can get the number of PUTs/GETs. Actually, the space covers those topologies gen-

erated by conventional mesh-based methods. And it can be proven that supposing N can be decomposed as the product of p prime factors, the space size $SS = \sum_{j=1}^{p} \sum_{i=1}^{j} \frac{(-1)^{j-i} i^p}{i!(j-i)!}$ (e.g., $SS = 115975$ when $p = 10$ and the detailed proof in §A.1). Such selectivity greatly facilitates seeking the most appropriate topology for an analytics job. Later, we will detail how to select the appropriate topology from the space, by carefully setting D.

Note that our approach may not work well in some corner cases, especially under prime function parallelism N. However, this problem can be easily addressed by allowing slight adjustments to the number N within a given bound, i.e., $[N-\alpha, N+\alpha]$. In our paper, we select $\alpha = 3$, and we will provide a detailed discussion on the impact of α in §4.5.

Last, compared to the mesh-based approach, the applicability gets significantly improved as well. As depicted in Figure 5(b), our approach even works for an asymmetric number of senders and receivers ($\#mapper = 8 \neq \#reducer = 6$), provided we keep the intermediate *flevel* the same size as the Reduce *flevel*, and link functions as Equation (2) suggests.

Candidates for Optimal Topology. For the Topology Optimizer, not provided with essential information (like function scheduling plan, data transmission manner, and other runtime states) to predict resulting completion time precisely, simply deciding the best topology by completion time is a rub. On the other hand, indiscriminately outputting all possible topologies forces all of them to go through all modules, incurring high overhead. Thus we adopt a middle-ground solution, i.e., to first select a small set of candidates, based solely on a comparison between their topological structure, then relegate the ultimate decision-making for the best to subsequent modules. In particular, though it's hard to directly find a total order for topologies' structure, comparison between them can be summarized as the following cases:

- Case 1. Under the same L, the topology with the fewest edges has the shortest completion time and data passing cost, as it transfers the data with the fewest PUTs/GETs that are more promptly processed by remote storage service.
- Case 2. Under different L, the comparison could be ambiguous, since on the one hand, larger L reduces edges, thus the number of PUTs/GETs. On the other hand, it transmits the intermediate data L times, potentially throttled by the function's network bandwidth.

Therefore, based on the partially ordered comparison, we add the locally optimal topology under each possible L, i.e., the one with the fewest edges, to our candidate set. Recall that the number of edges is $N \times \sum_{i=0}^{L-1} d_i$. Then suppose N can be decomposed into p prime factors, which means feasible L lies in $[1, p]$, candidate selection can be transformed into a series of optimization problems as follows:

$$\text{For } L \in [1,p], \quad \begin{cases} \text{minimize} & N \times \sum_{i=0}^{L-1} d_i \\ \text{subject to} & \prod_{i=0}^{L-1} d_i = N,\ d_i \in \mathbb{N}^+ \backslash \{1\} \end{cases} \tag{3}$$

We propose a dynamic programming algorithm to solve these problems at once. Let $MinSum(i,j)$ denote the minimized sum of factors when factorizing i into j factors. Then we need to find $MinSum(N,L)$ for $L \in [1,p]$. The state transition equation is as follows:

$$MinSum(i,j) = \begin{cases} \min_{n|i}(n + MinSum(i/n, j-1)) & j > 1 \\ i & j = 1 \end{cases} \tag{4}$$

As we see, the equation formulates the value of $MinSum(N,L)$ recursively in terms of its sub-problems. Thus we employ a bottom-up dynamic programming approach, i.e., iteratively solving $MinSum(i,j)$ with smaller i and j first and use their solutions to arrive at solutions to bigger i and j. More specifically, we can calculate all $MinSum(N,L)$ for $L \in [1,p]$ in a nested loop. In the inner loop, i starts from 1 to N, while in the outer loop, j progresses from 1 to p. Along the way, all desired $MinSum(N,L),\ 1 \leq L \leq p$ gets solved. Moreover, we use $Sol(i,j)$ to track the decomposition path of $MinSum(i,j)$, i.e., $Sol(i, j>1) =$ the selected n of $MinSum(i,j)$ in Equation (4) and $Sol(i,1) = i$. Then by iteratively putting $Sol(i,j)$ along the decomposition path of $MinSum(N,L)$ into a sequence, we can get the desired $D = \{d_i | 0 \leq i \leq L-1\}$, by which we can easily derive the corresponding L-level topology with the fewest edges.

3.3 Function Scheduler

The Function Scheduler assigns a scheduling plan to each of the candidate topologies, indicating when and on which worker each function should be invoked. While the "when" question is straightforward to deal with by monitoring the completion time of functions and following the data dependency between functions, the latter "where" question must be treated carefully to satisfy several important and interacting requirements. Next, we first formulate the problem and then demonstrate how to solve it.

Problem Formulation. The function placement problem is equivalent to partitioning the whole DAG into sub-graphs, where functions within each sub-graph must be co-located to pass data via local memory, while different sub-graphs are placed independently and communicate via remote storage. Then our goal is to search for a partitioning scheme that satisfies the following requirements:

1) *Traffic localization.* Since functions within sub-graphs are co-located and communicate via faster local memory, the resulting sub-graphs should include edges in the DAG as much as possible, to localize more traffic and thus accelerate the data passing.
2) *Transmission straggler avoidance.* Due to the synchronization barrier of the BSP model, the duration of each *clevel*'s transmission is decided by the slowest edge. Thus edges in the same *clevel* should be either all included in sub-graphs or not included at all, to avoid the benefit of faster local memory being offset by stragglers caused by remote storage.

3) *Load balancing.* The DAG must be partitioned until all subgraphs width, i.e., the number of functions in the *flevel* with the most functions, must be capped to ensure functions' computing and communication load can be easily spread among workers at all *flevels* and *clevels*.

Interleaved Graph Partitioning. As §2.4 suggests, the above requirements are contradictory in the original single-level and all-to-all topology. Surprisingly, the progressively converging *ML-Shuffle* brings an opportunity to achieve them simultaneously. Since it transforms the rigid all-to-all connection into multiple levels of more sparse connections, each *clevel* has the favorable feature as follows.

Theorem 1. *For a multi-level topology generated by the progressively converging method (the parallelism is N), the i-th level of links corresponding to factor d_i, along with two adjacent flevels of functions, can be divided into $\frac{N}{d_i}$ disjoint complete bipartite graphs with width d_i.*

Proof. According to Equation (2), $F_{i,m}$ and $F_{i,n}$, where $\lfloor \frac{m}{s_{i+1}} \rfloor = \lfloor \frac{n}{s_{i+1}} \rfloor$ *and* $m \equiv n (mod\ s_i)$, have the same receiver functions R. Hence, the functions at *flevel i* can be categorized into $\frac{N}{d_i}$ conjugacy classes, with the elements within each class sharing the same R. Each conjugacy class at *flevel i* and its R at *flevel* $i+1$ together constitute a complete bipartite graph with width d_i (detailed proof in § A.2). □

From Theorem 1, any *clevel* can be decomposed into isolated Complete Bipartite Graphs (CBGs), which are ideal units for function co-location, since all edges in the *clevel* are evenly included by same-sized CBGs as shown in Figure 6. In other words, by putting functions within each CBG to the same worker, data transmission of the *clevel* can be done via workers' memory instead of remote storage, greatly accelerating data passing. Meanwhile, different CBGs can be placed arbitrarily, without the need to be co-located.

So far we've found an excellent way to place functions for each individual *clevel*, yet the method can't be directly generalized to function placement for the whole multi-level graph. Since the communication of adjacent *clevels* involves a shared *flevel*, e.g., *clevels* 0 and 1 both involve *flevel* 1 (see Figure 6), co-location constraints of two *clevels* must be met at once, which leads to multiplied width of co-location units. Worse yet, when jointly considering all *clevels*, due to the cascade effect, functions in the whole graph must be co-located to the same worker, violating the load balance requirement. To address the problem, we employ an interleaved partitioning strategy, to decouple the tightly bound *clevels* so as to solve them independently. Specifically, it removes edges in all odd-numbered *clevels*, delegating that portion of data passing to remote storage. The rationality lies in the following corollary, which could be easily derived from Theorem 1.

Corollary 1. *For a k-level topology generated by progressively converging method (the parallelism is N), where each level has the corresponding factors d_0, d_1, ..., d_{k-1}, if we remove edges of all odd-numbered levels (1,3, ..., $\lfloor \frac{k}{2} \rfloor \times 2 - 1$), the whole graph can be divided into disjoint CBGs with width lying within D_{even} = {d_0,d_2, ..., $d_{\lfloor \frac{k-1}{2} \rfloor \times 2}$}*

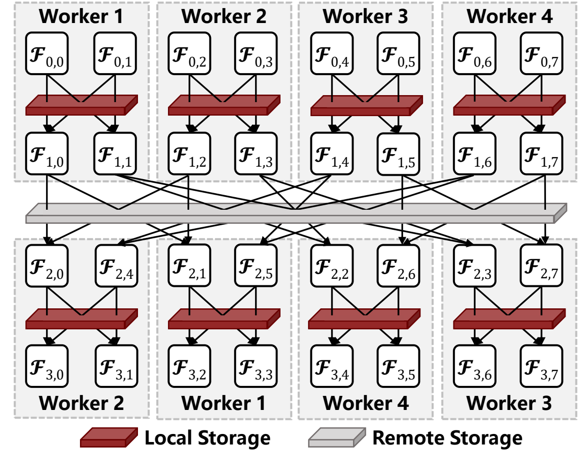


Figure 6: Function Scheduling.

The interleaved approach (see Figure 6) acts as a heuristic algorithm to solve the NP-hard graph partitioning problem [17,24,25], by giving quick solutions that fit the aforementioned requirements. Specifically, due to at least half of the *clevels* being left for local memory to perform data passing, performing function placement in units of resulting CBGs localizes over 50% of overall traffic. Meanwhile, since transmission media (i.e., local memory or remote storage) is assigned in an interleaved manner, no communication straggler exists during the job's execution. Last, it allows us to selectively decide which factors in D would be put in D_{even}, to minimize the resulting CBGs' width, thus achieving a fine-grained function placement that facilitates load balancing.

3.4 Configuration Modeler

Topology Optimizer (§3.2) has offered a group of candidate multi-level topologies, and Function Scheduler (§3.3) provided each with its appropriate function scheduling and placement scheme. Configuration Modeler's responsibility is to select the optimal one out of them. Since the additional function level of a multi-level topology only works to assist communication and doesn't change the job's computing time, Configuration Modeler opts to choose the one with the shortest overall data passing time.

To achieve this, the Configuration Modeler must precisely model each configuration's resulting passing time. As discussed in §2.4, during each *clevel*'s communication, either the function side or the storage side acts as the real bottleneck, depending on the actual number of requests and data volume. More specifically, the duration would be $T_i = 2 \times \max(T_{i,f}, T_{i,s}), 0 \leq i \leq L-1$, where $T_{i,f}$ and $T_{i,s}$ respectively represents the time spent on function putting/fetching

data in fixed rate and storage side processing received requests, and since the two parts are overlapping, we take the maximum of them. The reason behind the multiplier 2 is that each *clevel*'s communication includes sender functions writing to the storage side plus receivers reading back. Due to S3-based and memory-based communication alternating in different *clevels*, caused by the interleaved transmission media assignment (see §3.3), $T_{i,f}$ and $T_{i,s}$ are modeled differently in the two types of *clevels*. Suppose the function number is N in each *flevel*, in the S3-based *clevel* we have $T_{i,f} = \frac{D_i}{N*b_f}$ where D_i/N and b_f are each function's transmitted data volume at *flevel* i and bandwidth ceiling respectively, and $T_{i,s} = \frac{R_i}{q_s}$ where R_i is the involved number of requests at *clevel* i and q_s is S3's request rate. In contrast, for the memory-based *clevel* $T_{i,f} = \frac{D_i}{M*b_t}$ and $T_{i,s} = \frac{R_i}{M*q_t}$, where M is the number of cluster nodes, b_t and q_t respectively are the bandwidth ceiling and I/O rate limit of Tmpfs [1] which we leverage to establish the elastic reclaimed-memory file system. To summarize, the overall data transmission time of a multi-level network is:

$$T = 2 * \sum_{i=0}^{L-1} \begin{cases} \max(\frac{D_i}{N*b_f}, \frac{R_i}{q_s}), & i \text{ is odd}. \\ \max(\frac{D_i}{M*b_t}, \frac{R_i}{M*q_t}), & i \text{ is even}. \end{cases} \quad (5)$$

Except for the data volume D_i, other parameters in Equation (5) can be obtained before running the job. Yet D_i is only available by the runtime, preventing choosing the optimal configuration before the job runs. We use a sampling and profiling method to address the problem. As there is commonly a linear, or a non-linear but deterministic relationship between the size of input data and intermediate data [33], and D_i keeps consistent for all *clevels*, Configuration Modeler repeatedly samples the original input with different sizes and executes the job, recording the amount of intermediate data. Then each time it gets a new <input data size, intermediate data size> pair. By fitting these pairs using a curve, Configuration Modeler can estimate the intermediate data size under the whole input. Thus, by bringing all parameters into Equation (5), the Configuration Modeler predicts the transmission time of all candidate configurations, and selects the fastest one.

4 Evaluation

4.1 Experiment Setup

TestBed. We deploy our FaaS framework on 10 Amazon EC2 m6i.24xlarge instances, each with 96 vCPUs, 384GB memory, and 37.5 gigabits/s bandwidth, and we adopt Amazon S3 as the remote storage. All compute instances run Ubuntu 22.04 LTS with Linux kernel 5.15.0. For our FaaS framework, similar to FaasFlow, we run self-maintained functions within docker containers (24.0.6 version), rather than directly adopting function services that are not transparent to us, so as to better manage functions' execution and lifetime.

Workload. We adopt three widely employed benchmarks involving shuffle operations, ranging from typical MapReduce-style tasks to SQL-style queries.

- *TeraSort.* Sorting a dataset based on the specified key.
- *TPC-DS-Q16.* TPC-DS consists of multiple SQL queries. Among them, we select the most data-intensive one, i.e., the 16th query that performs a large joining via shuffle.
- *WordCount.* Counting word frequency in documents.

The datasets of the above workloads are respectively generated by Sort Benchmarks Generator [2], TPC-DS Tools [3], and Purdue MapReduce Benchmarks Suite [36].

Comparison. We compare MinFlow (denoted as MF) with the basic practice and two state-of-the-art works, in terms of both performance (execution time) and cost (fees charged).

- *Baseline.* The most common and straightforward approach, i.e., all intermediate results during shuffle are transferred through remote S3 object store, denoted as BL.
- *FaaSFlow.* FaaS framework with state-of-the-art function scheduling mechanism, which transmits intermediate data via local storage within workers, referred to as FF.
- *Lambada.* State-of-the-art topology optimizing method, performing multi-level shuffle to reduce PUTs/GETs to S3. We select its optimal configuration and denote it as LBD.

Configuration. During all our experiments, we set the resource limit of each function as 2 CPU, 3GB memory, and 75MB/s bandwidth, similar to prior research works [21, 26, 41, 47], to simulate a common setting of Amazon's commercial function service Lambda. By default, we respectively set the input size as 100GB and 200GB, and set the parallelism as 400 functions and 600 functions, since MinFlow mainly focuses on processing massive datasets with a large number of functions, which is in accordance with the serverless paradigm's goal to support hyper-scale computation with its superior scalability. Besides, we extensively adjust the input size and parallelism to show MinFlow's performance under broader settings (see §4.5).

4.2 Microbenchmark Results

Shuffle Time & Storage Cost. Now we evaluate MinFlow's effectiveness in improving the shuffle speed and saving the storage cost. Figure 7, 8 and 9 shows the shuffle time and normalized storage cost (divided by the BL's storage cost) of all approaches under three different workloads.

Taking TeraSort as the example (Figure 7, we first focus on the 600-function parallelism with 200GB input data size, the BL takes nearly 180s to finish the shuffle operation. During shuffling, not only do all functions await, but the bill for using functions continues increasing as function services usually charge based on time (e.g., in increments of 1ms). Besides, compared to BL, FF only slightly reduces the shuffle time and storage cost by 14.45% and 9.98%, respectively, since most of

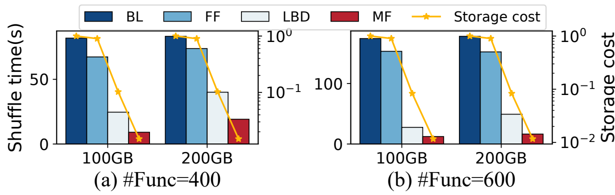


Figure 7: Shuffle Time of TeraSort.

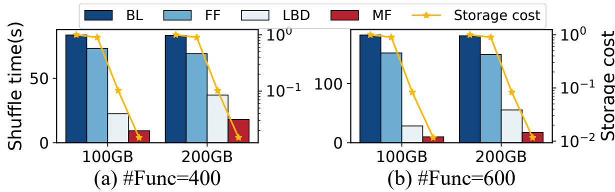


Figure 8: Shuffle Time of TPC-DS.

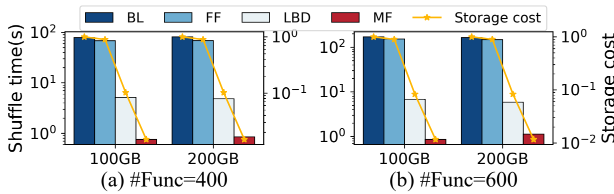


Figure 9: Shuffle Time of WordCount.

PUTs/GETs (324000 out of 360000 = 90%) carrying intermediate data still go through remote S3 and only a tiny portion (the rest 10%) can be performed via local storage, which we presented the reason in §2.4. In contrast, LBD could greatly accelerate the shuffle process with much less storage cost. Specifically, it shortens the shuffle time by over 72.36% and 67.69% compared to BL and FF, meanwhile saving the storage cost by 91.68% and 90.76%, respectively. Nevertheless, LBD still experiences ~50s idle time to wait for the shuffle's completion. As for MinFlow, it outperforms all the competitors in terms of performance by slashing the shuffle time to 16s. Compared to BL and FF, MinFlow achieves 10.8× and 9.3× shuffle acceleration respectively, and even compared to LBD, MinFlow still achieves 3× faster shuffle speed. In addition, MinFlow greatly saves the storage cost (98.84%, 98.71%, and 86% compared to BL, FF, and LBD), for it not only greatly reduces the number of PUTs/GETs but also largely eliminates additional intermediate data volume via local storage. Under 400-function TeraSort with 200GB input data size, similar to the 600-function parallelism, MinFlow preserves considerable performance and cost improvement – as Figure 7(a) shows it achieves 2.1× shuffle acceleration and 85.37% cost saving compared to LBD. Yet one noticeable change is that the performance benefit of MinFlow over BL and FF shrinks, although still reaches as high as 76.99% and 74.03%, respectively. It's because the performance degradation caused by excessive PUTs/GETs alleviates under lower parallelism. We will conduct more in-depth experiments about this phenomenon later (see §4.5).

Aside from the above TeraSort, Figure 8 and 9 show the results of TPC-DS-Q16 and WordCount. It can be found that while the results of TPC-DS-Q16 are quite similar to those of TeraSort, MinFlow shows much higher performance benefit over other approaches when running WordCount, as shown in Figure 9 where the vertical axis is logarithmic to more clearly present the shuffle time. For example, under 600 functions with 200GB input data, compared to BL and FF, MinFlow reduces the shuffle time by as high as 99.31% and 99.23%. Such phenomenon can be explained from the aspect of intermediate data size – while TPC-DS-Q16 and TeraSort share a common characteristic that the intermediate data size of shuffle is consistent with the input data size, WordCount has much less intermediate data since duplicate words in the input would be eliminated with a counter.

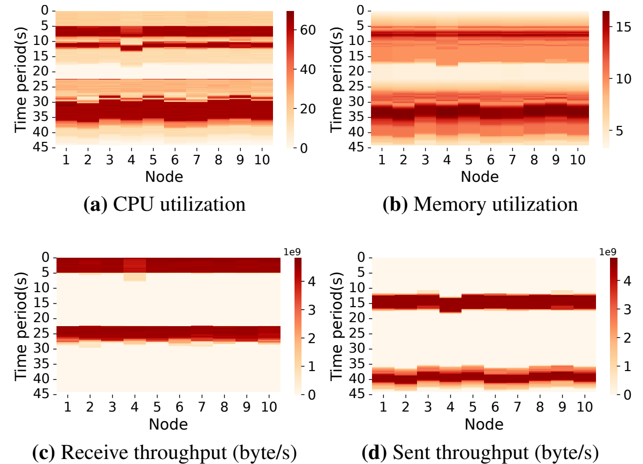


Figure 10: Load Balance. *TeraSort,600#F,200GB.*

Load Balance among Workers. Load balance has always been a necessity for large-scale distributed systems since it directly determines systems' resource efficiency and quality of service. To demonstrate MinFlow's capability in load balancing, we count the load of each worker every 50ms to show a fine-grained resource usage of workers. Figure 10(a), 10(b), 10(c) and 10(d) respectively show the CPU usage, memory occupation, and traffic load of all 10 workers when running TeraSort under 600 functions and 200GB input data with MinFlow, where the brighter red represents higher load. First, as we can see, all types of loads are kept even among workers throughout the process. Second, the load intensity of each worker varies noticeably along the timeline, which is in line with the BSP model's characteristic that compute/memory and traffic peaks appear alternatively. For example, the compute and memory peaks indicated by the bright-red "stripes" in Figure 10(a) and 10(b) represent the positive correlation between compute and memory peaks. On the other hand, the peaks of incoming traffic occur at around 0-5s (input) and 22.5-27.5s (GETs of remote storage shuffle), and peaks of outgoing traffic occur at around 12.5-17.5s (PUTs of remote storage shuffle) and 37.5-42.5s (output). They are both interleaved with the above compute/memory peak. In addition, by

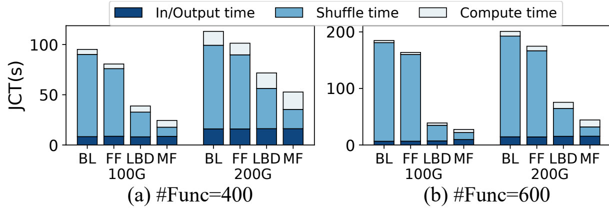


Figure 11: Overall Time of TeraSort.

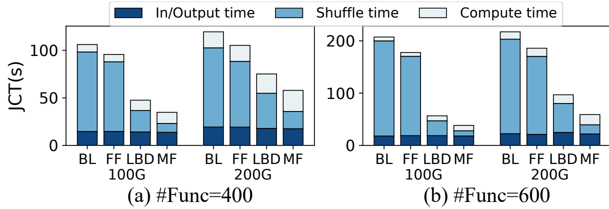


Figure 12: Overall Time of TPC-DS.

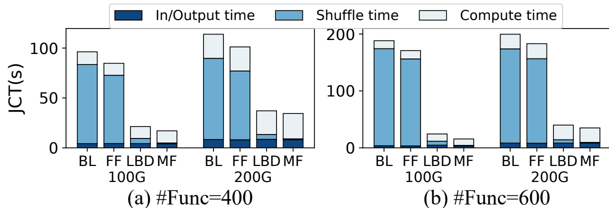


Figure 13: Overall Time of WordCount.

combining this set of results, we could find that for MinFlow the remote storage shuffle only accounts for ~10s out of the overall ~45s job running time, partly verifying MinFlow's high shuffle speed.

4.3 Overall Performance Analysis

Figure 11, 12 and 13 show the overall job completion time under the same setting as in §4.2. We still first take the 600-function with 200GB input data group as the example. In terms of the overall job completion time, as we can see, compared to other approaches MinFlow could contribute 41.35%-77.98% improvement for TeraSort workload, 39.12%-72.86% for TPC-DS, and 12.26%-82.46% for WordCount. The improvement mainly comes from shuffle time reduction, since among the compute, shuffle, and input/output time only the shuffle time changes significantly, while the other two parts basically remain constant across all approaches.

Among three workloads, for TeraSort and TPC-DS that have same-sized intermediate data with their input, shuffle time plays a non-neglectable part throughout all compared approaches. For example, in the TeraSort group BL, FF, and LBD respectively spend 88.65%, 87.23%, and 65.24% time on shuffling. By employing MinFlow the proportion could be reduced to 37.13%, contributing to not only more efficient computation but also better function use, since functions fees are charged in increments of time units, say 1ms. However, when it comes to WordCount, though the overall time improvement is still significant compared to BL and FF, the benefit over

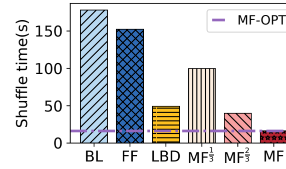


Figure 14: Breakdown.

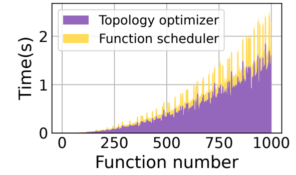


Figure 15: Scalability.

LBD shrinks, as Figure 13 shows. The reason is WordCount's reduction in intermediate size for word deduplication – While this does not noticeably impact BL and FF's overall time for their bottleneck lies in the excessive number of PUTs/GETs instead of the transmitted data volume, it greatly alleviates LBD's bottleneck that is mainly caused by function bandwidth limit. As a result, LBD's shuffle time only accounts for 14.83% of the overall job execution time, largely neutralizing the great shuffle time improvement of MinFlow over LBD.

Last, since theoretically the compute and input/output speed are proportional to the function number, increasing the parallelism can easily slash both compute and input/output time, which does not hold for shuffle time. Therefore shuffle time accounts for a higher portion under high parallelism, providing more optimization space. For example, as Figure 11 shows, under 200GB input data size, compared with 400-function parallelism in which MinFlow achieves 26.49%-53.38% overall time reduction, in 600-function parallelism the values increase to 41.35%-77.98% respectively. To conclude, these results demonstrate that MinFlow could improve the overall time considerably compared to existing works throughout all three workloads.

4.4 Breakdown and Overhead

Performance Breakdown. We progressively integrate the three components to show their respective contribution to MinFlow's shuffle time reduction. Figure 14 shows the results of TeraSort under 600 function and 200GB input data, where the MinFlow with only Topology Optimizer is referred to as $MF^{\frac{1}{3}}$, the version with both Topology Optimizer and Function Scheduler as $MF^{\frac{2}{3}}$, and the full version MinFlow denoted as MF. As it suggests, $MF^{\frac{1}{3}}$ decreases the shuffle time by 43.93% and 34.46% compared with BL and FF but is slower than LBD. This is because compared to LBD which offers a two-level shuffle, by default Topology Optimizer of MinFlow chooses the highest *clevel* number it can generate, to decrease entailed PUTs/GETs maximally. Yet this often incurs too much additional intermediate data. Fortunately, after the Function Scheduler is combined such issue gets greatly alleviated, thus $MF^{\frac{2}{3}}$ performs better than LBD, by 19% in the figure. Last, the full version of MinFlow further integrates Configuration Modeler to judiciously select the optimal *clevel* number and corresponding suitable function scheduling plan, instead of just gluing the Topology Optimizer and Function Scheduler. As a result, the full version of MinFlow could out-

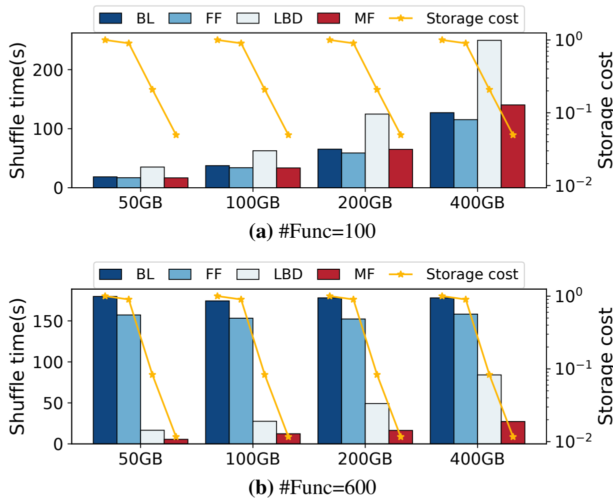


Figure 16: Various Input Data Size.

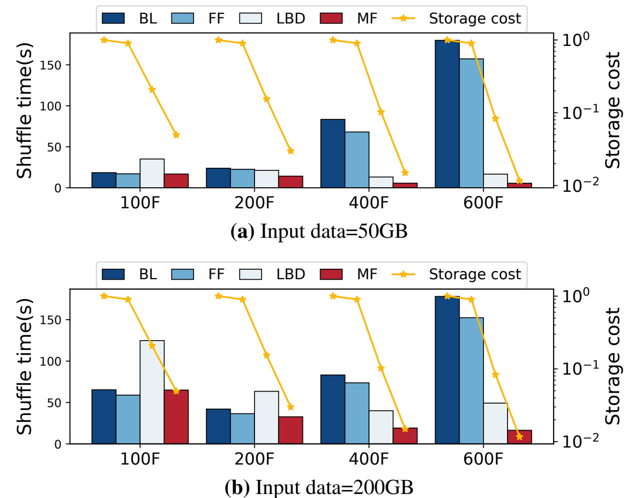


Figure 17: Tunable Function Parallelism.

perform other approaches by 66.62%-90.77%.

Besides, we further compare with MF-OPT, whose configuration is obtained by iteratively running all configurations and picking the one with the shortest shuffle time. As the purple line in Figure 14 shows, MinFlow achieves basically the same shuffle speed with MF-OPT once we ignore the tiny difference (below 1%) caused by our testbed's performance fluctuation.

System Overhead. Now we evaluate MinFlow's system overhead. First, the Topology Optimizer consumes additional CPU cycles to generate the candidate topologies before running jobs (see §3.2). Figure 15 presents the time cost, when MinFlow uses a single thread to perform topology calculation. As we can see, it basically increases linearly as the parallelism, i.e., the function number goes up. Under 600-function parallelism, the time is not above 1s, which can be ignored compared to MinFlow's improvement on shuffle time. Second, the Function Scheduler spends time searching in each of the candidate topologies for biparties, which are the basic function scheduling units (see §3.3). This part of the time cost is close to the topology calculation time as Figure 15 shows. As to some spikes in the figure, they appear when the parallelism value corresponds to more candidate topologies, i.e., the value that can be decomposed into more prime factors. For example, under $2\times2\times3\times5\times7 = 420$-function setting, it has 5 candidate topologies. In short, both of the above time costs are dwarfed by MinFlow's benefits. Moreover, if needed the time cost can be easily slashed by using multi-threads. Besides, though multi-level shuffle entails more functions, MinFlow eliminates the cost by keeping warm and reusing functions across levels. The memory consumed by local storage is also the reclaimed memory as in [25].

4.5 Impact of Different Configurations

As mentioned earlier, two factors impact MinFlow's performance, including the input size and function parallelism. Now we investigate the impact more extensively, by comparing MinFlow to other approaches under a broader range of input size and parallelism settings. For space limit, we only put results of TeraSort, while TPC-DS and WordCount show similar trends.

Input Size. First, we fix the parallelism as 100-function and tune the input size from 50GB to 100GB, 200GB, and 400GB. As we can see in Figure 16(a), across all approaches the shuffle time increases proportionally with the input size. The trend can be easily explained – Due to the number of PUTs/GETs to S3 not greater than $100\times100\times2$ = 20000, S3's speed of thousands of requests per second is enough to rapidly process them. Therefore for all approaches, the shuffle time is mainly determined by the volume of data to be transmitted, which is proportional to the input size. Note that even under such a low-parallelism setting, which is not MinFlow's target scenario, MinFlow could achieve near-optimal performance compared to others. By contrast, in the 600-function group (see Figure 16(b)), though LBD and MinFlow still exhibit a similar trend, the shuffle time of BL and FF remains consistent across all input sizes. Such difference stems from their distinct bottleneck. Specifically, for BL and FF the 600-function parallelism setting would incur $600\times600\times2$=720000 PUTs/GETs, making S3's speed the main bottleneck. As a result, their shuffle time is insensitive to the changing input size. However, due to LBD and MinFlow's great effectiveness in reducing the number of PUTs/GETs, their bottleneck still lies in functions' aggregated bandwidth sending/receiving intermediate data, leading to the shuffle time proportional to the input size. Note though it seems that under high-parallelism, say 600-function, the performance advantage of MinFlow over BL and FF shrinks as the input size increases, such trend would stop at a certain point where the input size is large enough to replace the massive PUTs/GETs as the new bottleneck.

Tunable Parallelism. Figure 17(a) shows the shuffle time results under parallelism of 100, 200, 400, and 600 functions, with 50GB input size. As we can see, for the low efficiency of performing shuffle, i.e., the huge number of PUTs/GETs

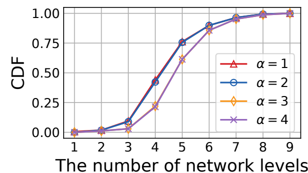


Figure 18: CDF of Network Levels for Prime within 1000.

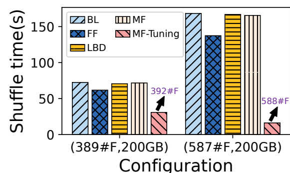


Figure 19: Shuffle Time under Prime Parallelism Cases.

to S3, both BL and FF's shuffle time keeps getting worse with the parallelism increasing, greatly impeding the critical scalability advantage of the serverless paradigm. By contrast, LBD exhibits a distinct trend that the shuffle time rapidly decreases as the parallelism gets higher, though its multiplied intermediate data, which must be transferred via remote S3, severely degrades its shuffle speed. For example, under low parallelism, say 100-function, it performs even worse than BL. In comparison, MinFlow not only preserves the continuously decreasing shuffle time but also avoids such degradation.

When the parallelism N is a prime number, we select the substitute in $[N-\alpha, N+\alpha]$ which can generate a network with as many *clevels* as possible. Figure 18 shows that after adjusting the prime numbers within 1000, the cumulative distribution of the number of network *clevels* can be generated. We can see that when $\alpha = 1$, all prime numbers except 2 can generate networks with more than two *clevels* and when $\alpha = 3$, more than 97% of prime numbers can generate networks with more than four *clevels*. Therefore, in cases involving prime parallelism, as illustrated in Figure 19, MinFlow after fine-tuning continues to outperform other approaches significantly.

Summary. MinFlow significantly outperforms other approaches in terms of both shuffle time and storage cost under high-parallelism, its target scenario. And even in a low-parallelism setting, it preserves good performance close to its best competitor, while considerably saving the storage cost.

5 Related Work

Optimization of Serverless DAGs. Several recent proposals have aimed to decrease job completion time by optimizing the performance of serverless DAGs. Orion [28] first proposes the idea of bundling multiple parallel invocations to mitigate execution skew and finds the best bundle size through trial and error. WiseFuse [29] goes a step further on Orion, it builds the performance model to determine bundle size and proposes the fusion of successive functions to reduce communication latency between consecutive stages in the DAG. However, given the huge data volume and dense topology inherent in serverless data analytics, fusion and bundling both struggle to mitigate the data exchange overhead. A complementary line of work provides efficient scheduling for serverless DAGs. Wukong [13] and FaaSFlow [25] provide decentralized and parallel scheduling distributed across function workers. Additionally, they harness over-provisioned local memory in the workers to expedite data exchange among functions within the same worker. This results in serverless DAGs that utilize both network I/O and local memory highly efficiently. Nevertheless, this approach proves inadequate when applied to serverless data analytics, as elaborated in §2.4. Overall, no prior work in this category can effectively reduce the data movement overhead of serverless data analytics.

Optimization of Serverless Intermediate Data Store. Besides DAGs optimization, recent work also reduces job completion time by optimizing the intermediate data store. Pocket [22] and Locus [35] show that current options for remote storage are either slow disk-based (e.g., S3) or expensive memory-based (e.g., ElastiCache). Thus, to balance performance and cost, Pocket combines different storage media (e.g., DRAM, NVMe, HDD) that users can choose to conform to their application needs. But this approach only makes economic sense when running different applications, e.g., when exclusively executing tasks like TeraSort, Pocket consistently selects the costly NVMe storage as the intermediate data repository. Faasm [37] and Cloudburst [38] accelerate data movement between functions, through a distributed shared memory across worker nodes. They rely on specific assumptions regarding the sandbox runtime and the programming interface exposed to tenants for developing their applications and in terms of consistency semantics and protocols between the FaaS workers and the backend storage. In contrast to existing efforts, MinFlow uses only cheap S3 and reclaimed memory, achieving performance and economic gains.

6 Conclusion

In this paper, we develop MinFlow, a holistic data passing framework for I/O-intensive serverless analytics jobs. MinFlow efficiently creates multi-level data passing topologies with fewer PUT/GET operations and uses an interleaved strategy to partition the topology DAG into complete bipartite sub-graphs. This optimizes function scheduling and cuts data transmission to remote storage by over one half. Additionally, MinFlow employs a precise model to pinpoint the best configuration. Experiments on our prototype demonstrate that MinFlow significantly outperforms state-of-the-art systems in both the job completion time and storage cost.

Acknowledgments

We thank our shepherd, Eno Thereska, and the anonymous reviewers for their comments. This work was supported in part by NSFC (62172382). The work of Yongkun Li was supported in part by the Youth Innovation Promotion Association CAS and Zhai Guanglong Scholarship. The work of John C.S. Lui was supported in part by the RGC GRF 14202923. Yongkun Li is USTC Tang Scholar.

A APPENDIX

A.1 Topology Space Size

Theorem A.1. *For a symmetric single-level shuffle network with function parallelism N, the topology space size of its multi-level shuffle network is* $SS = \sum_{j=1}^{p}\sum_{i=1}^{j}\frac{(-1)^{j-i}i^p}{i!(j-i)!}$*, where p refers to the number of prime factors of N.*

According to §3.2, the topology space size of the multi-level shuffle for the aforementioned network is equivalent to the search space size encompassing all factorizations of N. An intuitive way to explore all factorizations of N is to combine its prime factors, e.g., the 2-factorization of N is equivalent to dividing the prime factors of N into two nonempty sets. Next, we prove that this method results in the search space equal to SS in Theorem A.1.

Proof. Assume that $N = n_1 \times n_2 \times ... \times n_p$, where n_i is a prime, $1 \le i \le p$ and let $S(n,k)$ denote the number of $k-factorization$s of an integer with n prime factors. Then, $SS = \sum_{j=1}^{p} S(p, j)$. Note that we can derive all factorizations in $S(n,k)$ from factorizations in $S(n-1,k)$ and $S(n-1,k-1)$ through the following two methods:

- **case1:** Assume the extra factor of $S(n,k)$ compared to $S(n-1,k)$ as m. Combine m with any factor of a factorization in $S(n-1,k)$.
- **case2:** Assume the extra factor of $S(n,k)$ compared to $S(n-1,k-1)$ as m. Let m become a new factor of a factorization in $S(n-1,k-1)$.

Therefore, we can conclude that $S(n,k) = k \times S(n-1,k) + S(n-1,k-1)$ and $S(n,k)$ is the *Stirling Number of the Second Kind*, whose general formula is $\sum_{i=1}^{k}\frac{(-1)^{k-i}i^n}{i!(k-i)!}$ [42]. Furthermore, we can deduce that $SS = \sum_{j=1}^{p} S(p, j) = \sum_{j=1}^{p}\sum_{i=1}^{j}\frac{(-1)^{j-i}i^p}{i!(j-i)!}$. □

A.2 CBGs in Multi-level Networks

Theorem A.2. *For a multi-level topology generated by the progressively converging method (the parallelism is N), the i-th clevel of links corresponding to factor* d_i*, along with two adjacent flevels, can be divided into* $\frac{N}{d_i}$ *disjoint complete bipartite graphs with width* d_i*.*

Proof. Define the mapping function $\mathcal{F} := \langle f_1, f_2\rangle := \langle x, x\rangle \mapsto \langle \lfloor x/s_{i+1}\rfloor, x\%s_i\rangle, x \in \{0,1,...,N-1\}$. According to Equation (2), $F_{i,m}$ and $F_{i,n}$, where $\mathcal{F}(m) = \mathcal{F}(n)$, have the same receiver functions R.

We categorize the functions at *flevel* i into conjugacy classes, with the elements within each class sharing the same R. Assume that the ranges of $\mathcal{F}, f_1$ *and* f_2 are $\mathcal{R}, r_1$ *and* r_2 respectively, then $|\mathcal{R}| = |r_1| \times |r_2| = \frac{N}{s_{i+1}} \times s_i = \frac{N}{d_i}$, in other words, the functions at *flevel i* can be partitioned into $\frac{N}{d_i}$ conjugacy classes. For any element $\langle m,n\rangle$ in $\mathcal{R}$, we can find it's $\frac{s_{i+1}}{s_i} = d_i$ preimages, namely, $m \times s_{i+1} + k \times s_i + n, k \in \{0,1,...,d_i-1\}$. In summary, the functions at *flevel i* can be divided into $\frac{N}{d_i}$ conjugate classes of size d_i.

Because each conjugacy class at *flevel i* and its R at *flevel* $i+1$ together constitute a complete bipartite graph, the i-th *clevel* of links corresponding to factor d_i, along with two adjacent levels of functions, can be divided into $\frac{N}{d_i}$ disjoint complete bipartite graphs with width d_i. □

A.3 Applicability of Mesh-based Networks

Constructing k-level shuffle networks for function parallelism in N is equivalent to grouping N functions k times and performing intra-group shuffle after each grouping, which is revealed by Theorem A.2. Note that this process necessitates distinct groupings at each *flevel*.

Mesh-based method [32,34] groups functions by projecting N functions into a k-dimensional mesh to construct a k-level shuffle network. Specifically, they adopt the unary mapping $h_1(x,c) := x \mapsto \lfloor x/c\rfloor$ and $h_2(x,c) := x \mapsto x\%c$ for grouping, where $c|N, x \in \{0,1,...N-1\}$. However, this approach does not give guidance on selecting the side length (i.e., group size) of the k-dimensional mesh, while we conduct a detailed theoretical analysis in §3.2. Even worse, as shown in Theorem A.3, despite having insights into selecting an appropriate group size, the grouping methods outlined in [32, 34] still are proved to be ineffective in many cases.

Theorem A.3. *For multi-level networks where the number of flevels with the same group size is greater than three, unable to generate distinct groupings for flevels with the same group size using the unary mapping in* $h_1(x,c) := x \mapsto \lfloor x/c\rfloor$ *and* $h_2(x,c) := x \mapsto x\%c$*, where* $c|N, x \in \{0,1,...N-1\}$*.*

Proof. Assume that $N = s^n, n \ge 3$. According to §3.2, performing progressively converging for the N functions, we can construct a multi-level network where the number of *flevels* with group size s is greater than three. However, we can not use $h_1(x,c)$ and $h_2(x,c)$ to achieve this.

For $h_1(x,s)$, we can only perform $h_1(x,s) := x \mapsto \lfloor x/s\rfloor$ to divide N functions into groups with group size s. This is because that $h_1(x,s^m), 2 \le m \le n$ divides N functions into groups with group size greater s.

Likewise, as to $h_2(x,c)$, we can only perform a grouping $h_2(x,s^{n-1}) := x \mapsto x\%s^{n-1}$ distinct with $h_1(x,s)$ to divide N functions into groups with group size s. □

Corollary A.3. *For multi-level networks where the number of flevels with the same group size is greater than three, unable to generate distinct groupings for flevels with the same group size using arbitrary nesting of the unary mapping in* $\{h_1(x,c)|\, c|N\}\bigcup\{h_2(x,c)|\, c|N\}$*, where* $x \in \{0,1,...N-1\}$*.*

Proof. It is omitted as it is similar to the proof of Theorem A.3. □

Note that our method uses binary mapping $\mathcal{F}$ in Theorem A.2 instead of unary mapping to solve the limitations of mesh-based methods.

B Artifact Appendix

Abstract

Our artifact includes the prototype implementation of MinFlow and three other state-of-the-art comparison methods, along with the three data analytics benchmarks evaluated in our experiments. Additionally, we provide experiment scripts for reproducing our results on Amazon EC2 instances.

It's important to note that reproducing all of our results will take tens of hours and thousands of dollars with the Amazon cloud service.

Scope

The artifact has two main goals: The first is to enable the validation of the main claims presented in the paper. The second is to facilitate others in building upon MinFlow for their own projects. We include code to reproduce Figures 7-19.

Contents

The artifact is hosted in a git repository. This repository includes MinFlow 's source code as well as documentation and example applications. It is structured as follows:

benchmark/: This folder contains the code for the three evaluation applications (Terasort, TPC-DS, and WordCount) and input data generators for each application. By running `create_image.bash` in each application directory, serverless job-specific images can be deployed on the worker node.

config/: This folder contains the configuration file `config.py`, allowing users to configure the database and node information. It also provides options to select the application and comparison method for evaluation.

scripts/: This folder contains scripts (`conda_install.bash`, `python_install.bash`, and `docker_install.bash`) to automatically install the software dependencies of MinFlow, including Anaconda, Python, and Docker.

src/: This folder contains the source code of MinFlow and three other comparison methods (Baseline, FaaSFlow, and Lambada). We have integrated them and users can switch between different systems using the configuration file. The source code is structured as follows:

- **base/ & container/:** These two folders contain the code that builds the base images which expose hybrid-store APIs used in Function Scheduler (§3.3) and Configuration Modeler (§3.4) for applications.
- **parser/:** This folder contains the code that parses the application's YAML configuration file, written in the Workflow Definition Language, into a DAG object used in Topology Optimizer (§3.2).
- **grouping/:** This folder contains the code for Topology Optimizer (§3.2), Function Scheduler (§3.3), and Configuration Modeler (§3.4) to find the optimal execution plan.
- **workflow_manager/:** This folder contains the code for workflow management, including monitoring function status and triggering functions.
- **function_manager/:** This folder contains the code for managing containers (including creating, keeping warm, and removing) and executing functions.

test/: This folder contains the code for reproducing most of our evaluation results in Figures 7-9,11-13,16-17 (see folders **fast/** and **cost/**), 10 (see folder **load_balance/**), 14 (see folder **breakdown/**), 15 (see folder **scalability/**), 18 (see folder **alpha/**) and 19 (see folder **prime/**).

README.md: This documentation details how to install the software, set up the system, and reproduce the results in our paper.

Hosting

MinFlow artifact repository is hosted on GitHub and archived using Zenodo with a permanent DOI.

- **Repository:** https://github.com/lt2000/MinFlow.
- **Zenodo Archive:** https://zenodo.org/records/10494631.
- **DOI:** https://zenodo.org/doi/10.5281/zenodo.10494631.

Requirements

The artifacts have been developed and tested on an Amazon EC2 cluster comprising 10 m6i.24xlarge instances. And, the artifact uses Docker containers to host serverless functions, orchestrate the functions, and organize them in a DAG. §4.1 details the exact environment we used in our experiments.

Environment Setup

1. First, install the software dependencies by running the scripts directory in **scripts/** and mount the Tmpfs as our local storage.
2. Then, generate input data and build base and job-specific images for evaluation applications.
3. Configure the system configuration files and use `src/grouping/metadata.py` to generate the optimal execution plan.
4. Run the system and reproduce the results following the detailed instructions in `README.md`.

References

[1] Linux tmpfs, 2023. https://www.kernel.org/doc/html/latest/filesystems/tmpfs.html, Last accessed on 2023-6-29.

[2] Sort Benchmarks, 2023. https://sortbenchmark.org/, Last accessed on 2023-6-29.

[3] TPC-DS, 2023. https://www.tpc.org/tpcds/#, Last accessed on 2023-6-29.

[4] AMAZON. Amazon elasticache, 2023. https://aws.amazon.com/elasticache/, Last accessed on 2023-6-29.

[5] AMAZON. Amazon Lambda, 2023. https://aws.amazon.com/lambda, Last accessed on 2023-6-11.

[6] AMAZON. Amazon Lambda price, 2023. https://aws.amazon.com/cn/lambda/pricing/.

[7] AMAZON. Amazon Lambda usecases, 2023. https://docs.aws.amazon.com/lambda/latest/dg/applications-usecases.html, Last accessed on 2023-7-2.

[8] AMAZON. Amazon S3, 2023. https://aws.amazon.com/s3/, Last accessed on 2023-6-11.

[9] AMAZON. Amazon S3 price, 2023. https://aws.amazon.com/cn/s3/pricing/.

[10] AMAZON. Amazon S3 QPS limit, 2023. https://aws.amazon.com/cn/about-aws/whats-new/2018/07/amazon-s3-announces-increased-request-rate-performance/, Last accessed on 2023-6-29.

[11] Apache. Apache Kvrocks, 2023. https://github.com/apache/kvrocks, Last accessed on 2023-7-2.

[12] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 13–24, 2019.

[13] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng. Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 1–15, 2020.

[14] M. Chowdhury and I. Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 31–36, 2012.

[15] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[16] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection networks*. Morgan Kaufmann, 2003.

[17] J. Herrmann, M. Y. Özkaya, B. Uçar, K. Kaya, and U. V. Çatalyürek. Multilevel algorithms for acyclic partitioning of directed acyclic graphs. *SIAM J. Sci. Comput.*, 41(4):A2117–A2145, jan 2019.

[18] IBM. IBM cloud functions usecases, 2023. https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-use_cases, Last accessed on 2023-7-2.

[19] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[20] A. Khandelwal, Y. Tang, R. Agarwal, A. Akella, and I. Stoica. Jiffy: Elastic far-memory for stateful serverless analytics. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 697–713, 2022.

[21] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi. Understanding ephemeral storage for serverless analytics. In *2018 USENIX annual technical conference (USENIX ATC 18)*, pages 789–794, 2018.

[22] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI*, pages 427–444, 2018.

[23] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing*, volume 110. Benjamin/Cummings Redwood City, CA, 1994.

[24] H. R. Lewis. Michael r. πgarey and david s. johnson. computers and intractability. a guide to the theory of np-completeness. wh freeman and company, san francisco1979, x+ 338 pp. *The Journal of Symbolic Logic*, 48(2):498–500, 1983.

[25] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo. Faasflow: Enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 782–796, 2022.

[26] Y. Liu, B. Jiang, T. Guo, Z. Huang, W. Ma, X. Wang, and C. Zhou. Funcpipe: A pipelined serverless framework for fast and cost-efficient training of deep learning models. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(3):1–30, 2022.

[27] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi. Sonic: Application-aware data passing for chained serverless applications. In *USENIX Annual Technical Conference (USENIX ATC)*, 2021.

[28] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi. {ORION} and the three rights: Sizing, bundling, and prewarming for serverless

{DAGs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, 2022.

[29] A. Mahgoub, E. B. Yi, K. Shankar, E. Minocha, S. Elnikety, S. Bagchi, and S. Chaterji. Wisefuse: Workload characterization and dag transformation for serverless workflows. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(2):1–28, 2022.

[30] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. 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.

[31] Microsoft. Azure Functions usecases, 2023. https://learn.microsoft.com/en-us/dotnet/architecture/serverless/serverless-business-scenarios, Last accessed on 2023-7-2.

[32] I. Müller, R. Marroquín, and G. Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 115–130, 2020.

[33] S. M. Nabavinejad, M. Goudarzi, and S. Mozaffari. The memory challenge in reduce phase of mapreduce applications. *IEEE Transactions on Big Data*, 2(4):380–386, 2016.

[34] M. Perron, R. Castro Fernandez, D. DeWitt, and S. Madden. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 131–141, 2020.

[35] Q. Pu, S. Venkataraman, and I. Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *NSDI*, volume 19, pages 193–206, 2019.

[36] Purdue. Purdue mapreduce benchmarks suite, 2023. https://engineering.purdue.edu/~puma/datasets.htm, Last accessed on 2023-6-29.

[37] S. Shillaker and P. Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433, 2020.

[38] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[39] S. Thomas, L. Ao, G. M. Voelker, and G. Porter. Particle: ephemeral endpoints for serverless networking. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 16–29, 2020.

[40] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[41] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, 2018.

[42] Wolfram. Stirling Number of the Second Kind, 2023. https://mathworld.wolfram.com/StirlingNumberoftheSecondKind.html.

[43] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. 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, 2012.

[44] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[45] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M. J. Freedman. Riffle: Optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[46] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica. Caerus: Nimble task scheduling for serverless analytics. In *NSDI*, pages 653–669, 2021.

[47] J. Zhang, A. Wang, X. Ma, B. Carver, N. J. Newman, A. Anwar, L. Rupprecht, V. Tarasov, D. Skourtis, F. Yan, and Y. Cheng. Infinistore: Elastic serverless cloud storage. *Proc. VLDB Endow.*, 16(7):1629–1642, may 2023.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

COLE: A Column-based Learned Storage for Blockchain Systems

Ce Zhang*, Cheng Xu*, Haibo Hu†, Jianliang Xu*

Hong Kong Baptist University †Hong Kong Polytechnic University

Abstract

Blockchain systems suffer from high storage costs as every node needs to store and maintain the entire blockchain data. After investigating Ethereum's storage, we find that the storage cost mostly comes from the index, i.e., Merkle Patricia Trie (MPT). To support provenance queries, MPT persists the index nodes during the data update, which adds too much storage overhead. To reduce the storage size, an initial idea is to leverage the emerging learned index technique, which has been shown to have a smaller index size and more efficient query performance. However, directly applying it to the blockchain storage results in even higher overhead owing to the requirement of persisting index nodes and the learned index's large node size. To tackle this, we propose COLE, a novel column-based learned storage for blockchain systems. We follow the column-based database design to contiguously store each state's historical values, which are indexed by learned models to facilitate efficient data retrieval and provenance queries. We develop a series of write-optimized strategies to realize COLE in disk environments. Extensive experiments are conducted to validate the performance of the proposed COLE system. Compared with MPT, COLE reduces the storage size by up to 94% while improving the system throughput by 1.4×-5.4×.

1 Introduction

Blockchain, as the backbone of cryptocurrencies and decentralized applications [38,52], is an immutable ledger built on a set of transactions agreed upon by untrusted nodes. It employs cryptographic hash chains and consensus protocols for data integrity. Users can retrieve historical data from blockchain nodes with integrity assurance, also known as provenance queries. However, all nodes are required to store the complete transactions and ledger states, leading to amplified storage expenses, particularly as the blockchain continues to grow. For example, the Ethereum blockchain requires about 16TB storage as of December 2023, with an annual growth of around 4TB [1]. This storage requirement may compel the resource-limited nodes to retain only the data of a few recent blocks, which restricts the ability to support data provenance. The nodes that maintain the complete data may also leave the network due to the rapidly increasing storage size, which potentially affects system security.

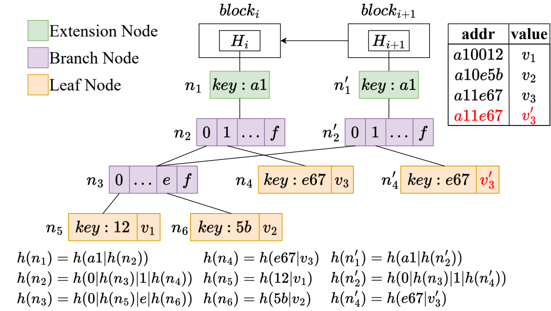


Figure 1: An Example of Merkle Patricia Trie

To tackle the storage issue, we investigate Ethereum's index, Merkle Patricia Trie (MPT), to identify the storage bottleneck. MPT combines Patricia Trie with Merkle Hash Tree (MHT) [37] to ensure data integrity. During data updates, its index nodes are persisted to support provenance queries. Figure 1 shows an example of an MPT storing three state addresses across two blocks. Each node is augmented with a digest from its content and child nodes (e.g., $h(n_1) = h(a1|h(n_2))$). The root hash secures data integrity through the collision-resistance of the cryptographic hash function and the hierarchical structure. With each new block, MPT retains obsolete nodes from the preceding block. For example, in block $i+1$, updating address $a11e67$ with v_3' introduces new nodes n_1', n_2', n_4', while old nodes n_1, n_2, n_4 endure. This setup allows historical data retrieval from any block (e.g., for address $a11e67$ in block i, value v_3 is retrieved by traversing nodes n_1, n_2, and n_4).

However, this approach adds too much storage overhead due to duplicating nodes along the update path (e.g., n_1, n_2, n_4 and n_1', n_2', n_4' in Figure 1). Consequently, most storage overhead comes from the index rather than the underlying data. In a preliminary experiment with 10 million transactions under the SmallBank workload [17], we observed that the underlying data contributes only 2.8% of the total storage. Thus, a more compact index supporting data integrity and provenance queries is imperative.

Recently, a novel indexing technique, learned index [15,20, 26,54], has emerged and shows notably smaller index size and faster query speed. The improved performance comes from the substitution of the directing keys in index nodes with a learned model. For instance, consider a key-value database with linear key distribution: $(1,v_1),(2,v_2),\cdots,(n,v_n)$. In a

traditional B+-tree with fanout f, this leads to $O(\frac{n}{f})$ nodes and $O(\log_f n)$ levels, resulting in $O(n)$ storage costs and $O(\log_f n \cdot \log_2 f)$ query times. Conversely, using a simple linear model $y = x$ enables accurate data positioning with just $O(1)$ storage and $O(1)$ query times. Although this example may not perfectly reflect real-world applications, it highlights that the learned index outperforms traditional indexes significantly when the model effectively learns the data.

In view of the advantages of the learned index, one may want to apply it to blockchain storage to improve performance. However, the current learned indexes do not support both data integrity and provenance queries required by blockchain systems. A naive approach is to combine the learned index with MHT [37] and make the index nodes persistent, as in MPT. Nonetheless, this is not feasible due to the larger node size of the learned index. The fanout of such a node is mainly dictated by data distribution. In favorable cases, only a few models are needed to index data, leading to a node fanout comparable to data magnitude. Thus, persisting learned index nodes might incur even higher storage overhead than MPT. Our evaluation in Section 8 shows that a learned index with persistent nodes is 5× to 31× larger than MPT. Furthermore, as blockchain systems require durable disk-based storage and often involve frequent data updates, the learned index should be optimized for both disk and write operations. Therefore, a blockchain-friendly learned index needs to be proposed.

In this paper, we propose COLE, a novel column-based learned storage for blockchain systems that overcomes the limitations of current learned indexes and supports provenance queries. The key challenge in adapting learned indexes to blockchains is the need for node persistence, which may lead to substantial storage overhead. COLE tackles this issue with an innovative *column-based* design, inspired by column-based databases [4, 36]. In this design, each ledger state is treated as a "column", with different versions of a state stored contiguously and indexed using *learned models* within *the latest block's* index. This enables efficient data updates as append operations with associated version numbers (i.e., state's block heights). Moreover, historical data queries no longer traverse previous block indexes, but utilize the learned index in the most recent block. The column-based design also simplifies model learning and reduces disk IOs.

To handle frequent data updates and enhance write efficiency in COLE, we propose adopting the *log-structured merge-tree* (LSM-tree) [33,41] maintenance approach to manage the learned models. This involves inserting updates into an in-memory index before merging them into on-disk levels that grow exponentially. For each on-disk level, we design a disk-optimized learned model that can be constructed in a *streaming* way, which enables efficient data retrieval with minimal IO cost. To guarantee data integrity, we construct an m-ary complete MHT for the blockchain data in each on-disk level. The root hashes of the in-memory index and all MHTs combine to create a root digest that attests to the entire blockchain data. However, recursive merges during write operations can lead to long-tail latency in the LSM-tree approach. To alleviate this issue, we further develop a novel checkpoint-based asynchronous merge strategy to ensure the synchronization of the storage among blockchain nodes.

To summarize, this paper makes the following contributions:

- To the best of our knowledge, COLE is the first column-based learned storage that combines learned models with the column-based design to reduce storage costs for blockchain systems.
- We propose novel write-optimized and disk-optimized designs to store blockchain data, learned models, and Merkle files for realizing COLE.
- We develop a new checkpoint-based asynchronous merge strategy to address the long-tail latency problem for data writes in COLE.
- We conduct extensive experiments to evaluate COLE's performance. The results show that compared with MPT, COLE reduces storage size by up to 94% and improves system throughput by 1.4×-5.4×. Additionally, the proposed asynchronous merge decreases long-tail latency by 1-2 orders of magnitude while maintaining a comparable storage size.

The rest of the paper is organized as follows. We present some preliminaries about blockchain storage in Section 2. Section 3 gives a system overview of COLE. Section 4 designs the write operation of COLE, followed by an asynchronous merge strategy in Section 5. Section 6 describes the read operations of COLE. Section 7 presents a complexity analysis. The experimental evaluation results are shown in Section 8. Section 9 discusses the related work. Finally, we conclude our paper in Section 10.

2 Blockchain Storage Basics

In this section, we give some necessary preliminaries to introduce the proposed COLE. Blockchain is a chain of blocks that maintains a set of states and records the transactions that modify these states. To establish a consistent view of the states among mutually untrusted blockchain nodes, a consensus protocol is utilized to globally order the transactions [7, 38, 45]. The transaction's execution program is known as *smart contract*. A smart contract can store states, each of which is identified by a state address $addr$. In Ethereum [52], both the state address $addr$ and the state value $value$ are fixed-sized strings. Figure 2 shows an example of the block data structure. The header of a block consists of (i) H_{prev_blk}, the hash of the previous block; (ii) TS, the timestamp; (iii) π_{cons}, the consensus protocol related data; (iv) H_{tx}, the root digest of the transactions in the current block; (v) H_{state}, the root digest of the states. The block body includes the transactions, states, and their corresponding Merkle Hash Tree (MHTs).

MHT is a prevalent hierarchical structure to ensure data

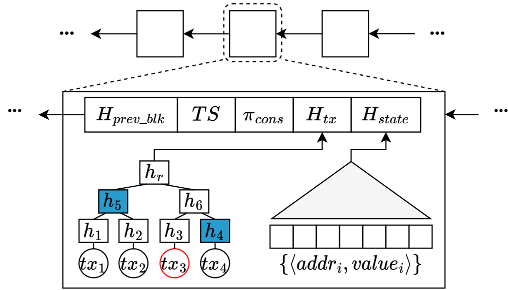


Figure 2: Block Data Structure

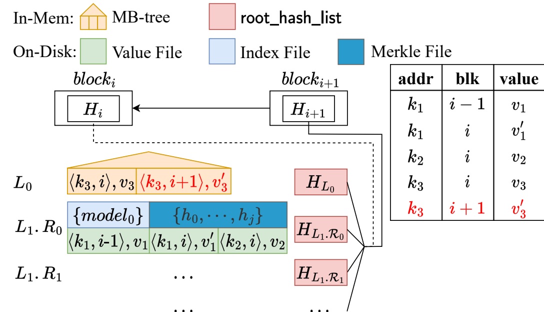


Figure 3: Overview of COLE

integrity [37]. In the context of blockchain, MHT is built for the transactions of each block and the ledger states. Figure 2 shows an example of an MHT of a block's transactions. The leaf nodes are the hash values of the transactions (e.g., $h_1 = h(tx_1)$). The internal nodes are the hash values of their child nodes (e.g., $h_5 = h(h_1||h_2)$). MHT enables the proof of existence for a given transaction. For example, to prove tx_3, the sibling hashes along the search path (i.e., h_4 and h_5, shaded in Figure 2) are returned as the proof. One can verify tx_3 by reconstructing the root hash using the proof (i.e., $h(h_5||h(h(tx_3)||h_4)))$ and comparing it with the one in the block header (i.e., H_{tx}). Apart from being used in the blockchain, MHT has also been extended to database indexes to support result integrity verification for different queries. For example, MHT has been extended to Merkle B+-tree (MB-tree) by combining the Merkle structure with B+-tree, to support trustworthy queries in relational databases [29].

The blockchain storage uses an index to efficiently maintain and access the states [50, 52]. Besides the write and read operations that a normal index supports, the index of the blockchain storage should also fulfill the two requirements we mentioned before: (i) ensuring the *integrity* of the indexed blockchain states, (ii) supporting *provenance queries* that enable blockchain users to retrieve historical state values with integrity assurance. With these requirements, the index of the blockchain storage should support the following functions:

- `Put`$(addr, value)$: insert the state with the address *addr* and the value *value* to the current block;
- `Get`$(addr)$: return the *latest* value of the state at address *addr* if it exists, or returns *nil* otherwise;
- `ProvQuery`$(addr, [blk_l, blk_u])$: return the provenance query results $\{value\}$ and a proof π, given the address *addr* and the block height range $[blk_l, blk_u]$;
- `VerifyProv`$(addr, [blk_l, blk_u], \{value\}, \pi, H_{state})$: verify the provenance query results $\{value\}$ w.r.t. the address, the block height range, the proof, and H_{state}, where H_{state} is the root digest of the states.

Ethereum employs Merkle Patricia Trie (MPT) to index blockchain states. In Section 1, we have shown how MPT implements `Put`$(\cdot)$ and `ProvQuery`$(\cdot)$ using Figure 1 and the address $a11e67$. We now explain the other two functions using the same example. `Get`$(a11e67)$ finds $a11e67$'s latest value v_3' by traversing n_1', n_2', n_4' under the latest block $i+1$. After `ProvQuery`$(a11e67, [i, i])$ gets v_3 and the proof $\pi = \{n_1, n_2, n_4, h(n_3)\}$ in block i, `VerifyProv`$(\cdot)$ is used to verify the integrity of v_3 by reconstructing the root digest using the nodes from n_4 to n_1 in π and checks whether the reconstructed one matches the public digest H_i in block i and whether the search path in π corresponds to the address $a11e67$.

3 COLE Overview

This section presents COLE, our proposed column-based learned storage for blockchain systems. We first give the design goals and then show how COLE achieves these goals.

3.1 Design Goals

We aim to achieve the following design goals for COLE:

- **Minimizing storage size.** To scale up the blockchain system, it is important to reduce the storage size by leveraging the learned index and column-based design.
- **Supporting the requirements of blockchain storage.** As blockchain storage, it should ensure data integrity and support provenance queries as mentioned in Section 2.
- **Achieving efficient writes in a disk environment.** Since blockchain is write-intensive and all data needs to be preserved on disk, the system should be write-optimized and disk-optimized to achieve better performance.

3.2 Design Overview

Figure 3 shows the overview of COLE. Following the column-based design [4,36], we adopt an analogy between blockchain states and database columns. Each state's historical versions are contiguously stored in the index of the latest block. When a state is updated in a new block, the state and its version number (i.e., block height) are appended to the index where all of the state's historical versions are stored. For indexing historical state values, we use a *compound key* $\mathcal{K}$ in the form of $\langle addr, blk \rangle$, where *blk* is the block height when the value of *addr* was updated. In Figure 3, when block $i+1$ updates the state at address k_3 (highlighted in red), a new compound key of k_3, $\mathcal{K}_3' \leftarrow \langle k_3, i+1 \rangle$, is created. Then, the updated value v_3' indexed by $\mathcal{K}_3'$ is inserted into COLE. With the column-based design, v_3' is stored next to k_3's old version v_3. Compared with

the MPT in Figure 1, the cumbersome node duplication along the update path (e.g., n_1, n_2, n_4 and n'_1, n'_2, n'_4) is avoided to save the storage overhead.

To mitigate the high write cost associated with learned models for indexing blockchain data in a column-based design, we propose using the LSM-tree maintenance strategy in COLE. It structures index storage into levels of exponentially increasing sizes. New data is initially added to the first level. When the level reaches its pre-defined maximum capacity, all the data in that level is merged into a sorted run in the next level. This merge operation can occur recursively until the capacity requirement is no longer violated. The first level, often highly dynamic, is typically stored in memory, while other levels reside on disk. COLE employs Merkle B+-tree (MB-tree) [29] for the first level and disk-optimized learned indexes for subsequent levels. We choose MB-tree over MPT for the in-memory level due to its better efficiency in compacting data into sorted runs and flushing them to the first on-disk level.

Each on-disk level contains a fixed number of sorted runs, each of which is associated with a value file, an index file, and a Merkle file:

- **Value file** stores blockchain states as compound key-value pairs, which are ordered by their compound keys to facilitate the learned index.
- **Index file** helps locate blockchain states in the value file during read operations. It uses a disk-optimized learned index, inspired by PGM-index [20], for efficient data retrieval with minimal IO cost.
- **Merkle file** authenticates the data stored in the value file. It is an m-ary complete MHT built on the compound key-value pairs.

Note that since the model construction and utilization require numerical data types, we convert a compound key into a big integer using the binary representation of the address and the block height. For example, given a compound key $\mathcal{K} \leftarrow \langle addr, blk \rangle$, its big integer is computed as $binary(addr) \times 2^{64} + blk$, where blk is a 64-bit value. Moreover, to ensure data integrity, root hashes of both the in-memory MB-tree and the Merkle files of each on-disk run are combined to create a root_hash_list. The root digest of states, stored in the block header, is computed from this list. This list is cached in memory to expedite root digest computation.

With this design, to retrieve the state value of address $addr_q$ at a block height blk_q, a compound key $\mathcal{K}_q \leftarrow \langle addr_q, blk_q \rangle$ is employed. The process entails a level-wise search within COLE, initiated from the first level. The MB-tree or the learned indexes in other levels are traversed. The search ceases upon encountering a compound key $\mathcal{K}_r \leftarrow \langle addr_r, blk_r \rangle$ where $addr_r = addr_q$ and $blk_r \leq blk_q$, at which point the corresponding value is returned. For retrieving the latest value of a state, the procedure remains similar but with the search key set to $\langle addr_q, max_int \rangle$, where max_int is the maximum integer. That is, the search is stopped as long as a state value with the queried address $addr_q$ is found.

Algorithm 1: Write Algorithm

```
1  Function Put(addr, value)
     Input: State address addr, value value
2    blk ← current block height; K ← ⟨addr, blk⟩;
3    Insert ⟨K, value⟩ into the MB-tree in L0;
4    if L0 contains B compound key-value pairs then
5      Flush the leaf nodes in L0 to L1 as a sorted run;
6      Generate files F_V, F_I, F_H for this run;
7      i ← 1;
8      while Li contains T runs do
9        Sort-merge all the runs in Li to Li+1 as a new run;
10       Generate files F_V, F_I, F_H for the new run;
11       Remove all the runs in Li;
12       i ← i + 1;
13   Update H_state when finalizing the current block;
```

4 Write Operation of COLE

We now detail the write operation of COLE. As mentioned in Section 3.2, COLE organizes the storage using an LSM-tree, which consists of an in-memory level and multiple on-disk levels. The in-memory level has a capacity of B states in the form of compound key-value pairs. Once this capacity is reached, the in-memory level is flushed to the disk as a sorted run. Similarly, when the first on-disk level reaches its capacity of T sorted runs, they are merged into a new run in the next level. This merging process continues for subsequent disk levels, with the size of each run growing exponentially with a ratio of T. That is, level i has a maximum of $B \cdot T^i$ states.

Algorithm 1 shows COLE's write operation. It starts by calculating a compound key for the state using the address and the current block height (Line 2). The compound key-value pair is inserted into the in-memory level L_0 indexed by the MB-tree (Line 3). As L_0 fills up, it is flushed to the first on-disk level L_1 as a sorted run (Line 5). The value file $\mathcal{F}_V$ is generated by scanning compound key-value pairs in the MB-tree's leaf nodes (Line 6). At the same time, the index file $\mathcal{F}_I$ and the Merkle file $\mathcal{F}_H$ are constructed in a *streaming* manner (see Section 4.1, Section 4.2 for details). When on-disk level L_i fills up (i.e., with T runs), all the runs in L_i are merge-sorted as a new run in the next level L_{i+1}, with corresponding three files generated (Lines 8 to 11). This level-merge process continues recursively until a level does not fill up. The blockchain's state root digest H_{state} is computed by hashing the concatenation of the root hash of L_0's MB-tree and root hashes of runs in other levels, stored in root_hash_list, when finalizing the current block (Line 13).

Example. *Figure 4 shows an example of the insertion of s_{10}. For clarity, we show only the states and the value files but omit the index files and Merkle files. Assume $B = 2$ and $T = 3$. The sizes of the runs in L_1 and L_2 are 2 and 6, respectively. After s_{10} is inserted into in-memory level L_0, the level is full*

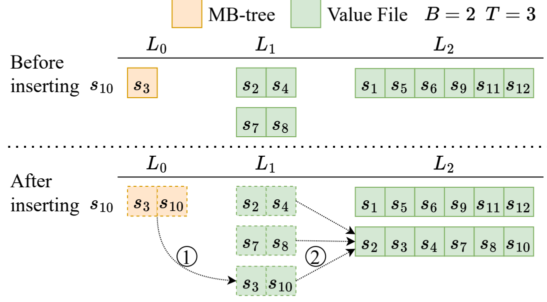


Figure 4: An Example of Write Operation

and its states are flushed to L_1 as a sorted run (step ①). This incurs L_1 reaching the maximum number of runs. Thus, all the runs in L_1 are next sort-merged as a new run, placed in L_2 (step ②). Finally, L_0 and L_1 are empty and L_2 has two runs, each of which contains six states.

A common optimization technique to speed up read operations is to integrate a Bloom filter into the in-memory MB-tree and each run in the on-disk levels. We incorporate the Bloom filter into COLE with careful consideration. First, they should be built upon the addresses of the underlying states rather than their compound keys to facilitate read operations. Second, since the Bloom filters may produce false positives, if they indicate that an address exists, we further resort to the normal read process of the corresponding MB-tree or the disk run to ensure the search correctness. We will elaborate on their usage during the read operation in Section 6. Moreover, the Bloom filters should be incorporated alongside the root hashes of each run when computing the states' root digest. This is needed to verify the result integrity during provenance queries.

4.1 Index File Construction

An index file consists of the models that can be used to locate the positions of the states' compound keys in the value file. Inspired by PGM-index [20], we start by defining an ε-bounded piecewise linear model (or *model* for short) as follows.

Definition 1 (ε-Bounded Piecewise Linear Model)**.** *The model is a tuple of $\mathcal{M} = \langle sl, ic, k_{min}, p_{max} \rangle$, where sl and ic are the slope and intercept of the linear model, k_{min} is the first key in the model, and p_{max} is the last position of the data covered by the model.*

Given a model, one can predict a compound key $\mathcal{K}$'s position p_{real} in a file, if $\mathcal{K} \geq k_{min}$. The predicted position p_{pred} is calculated as $p_{pred} = \min(\mathcal{K} \cdot sl + ic, p_{max})$, which satisfies $|p_{pred} - p_{real}| \leq \varepsilon$. Since files are often organized into pages, we set ε as half the number of models that can fit into a single disk page to generate the models in a disk-friendly manner. As will be shown, this reduces the IO cost by ensuring that at most two pages need to be accessed per model during read operations.

Algorithm 2: Learn Models from a Stream

1 **Function** `BuildModel`$(\mathcal{S}, \varepsilon)$
  Input: Input stream $\mathcal{S}$, error bound ε
  Output: A stream of models $\{\mathcal{M}\}$
2   $k_{min} \leftarrow \emptyset$, $p_{max} \leftarrow \emptyset$, $g_{last} \leftarrow \emptyset$;
3   Init an empty convex hull $\mathcal{H}$;
4   **foreach** $\langle \mathcal{K}, p_{real} \rangle \leftarrow \mathcal{S}$ **do**
5   **if** $k_{min} = \emptyset$ **then** $k_{min} \leftarrow \mathcal{K}$;
6   Add $\langle BigNum(\mathcal{K}), p_{real} \rangle$ to $\mathcal{H}$;
7   Compute the minimum parallelogram $\mathcal{G}$ that covers $\mathcal{H}$;
8   **if** $\mathcal{G}.height \leq 2\varepsilon$ **then**
9       $p_{max} \leftarrow p_{real}, g_{last} \leftarrow \mathcal{G}$;
10   **else**
11   Compute slope sl and intercept ic from g_{last};
12       $\mathcal{M} \leftarrow \langle sl, ic, k_{min}, p_{max} \rangle$;
13   **yield** $\mathcal{M}$;
14       $k_{min} \leftarrow \mathcal{K}$;
15   Init a new convex hull $\mathcal{H}$ with $\langle BigNum(\mathcal{K}), p_{real} \rangle$;

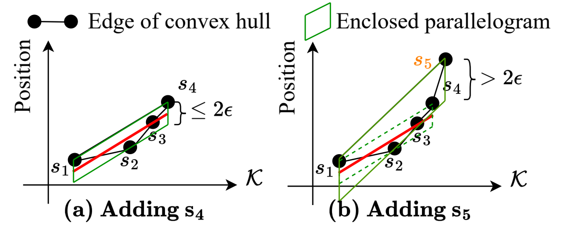


Figure 5: An Example of Model Learning

To compute models from a stream of compound keys and their corresponding positions, we treat each compound key and its position as a point's coordinates. Upon the arrival of a new compound key, we convert it into a big integer using the binary representation of the address and the block height as mentioned in Section 3.2. Next, we find the smallest convex shape containing all the existing input points, which is known as a convex hull. Note that this convex hull can be computed incrementally in a streaming fashion [40]. Then, we find the minimal parallelogram that covers the convex hull, with one side aligned to the vertical axis (i.e., the position axis). If the parallelogram's height stays under 2ε, all existing inputs can fit into a single model. In this case, we try to include the next compound key in the stream for model construction. However, if the current parallelogram fails to meet the height criteria, the slope and intercept of the central line in the parallelogram will be used to build a model that covers all existing compound keys except the current one. After this, a new model will be built, starting from the current compound key. We summarize the algorithm in Algorithm 2.

Example. *Figure 5 shows an example of model learning from a stream. Assume states s_1 to s_3 form a convex hull, with its minimal parallelogram satisfying the height criterion (i.e., below 2ε). After state s_4 is added, the parallelogram's height remains within 2ε (see Figure 5(a)), indicating that states s_1 to s_4 can be fit into one model. However, after the next state s_5 is added, the parallelogram's height exceeds 2ε (see*

Algorithm 3: Index File Construction

```
1  Function ConstructIndexFile(S, ε)
       Input: Input stream S of compound key-position pairs
       Output: Index file F_I
2      Create an empty index file F_I;
3      Invoke BuildModel(S, ε) and write to F_I;
4      n ← # of pages in F_I;
5      while n > 1 do
6          S ← {⟨M.k_min, pos⟩ | foreach ⟨M, pos⟩ ∈ F_I[−n :]};
7          Invoke BuildModel(S, ε) and append to F_I;
8          n ← # of pages in F_I − n;
9      return F_I;
```

Figure 5(b)). Thus, the slope and intercept of the previous parallelogram's central line (highlighted in red) are used to build a model for s_1 to s_4, with s_5 reserved for the next model.

Algorithm 3 shows the overall procedure of index file generation. During flush or sort-merge operations in Algorithm 1, ordered compound keys and state values are generated and written streamingly into the value file. Meanwhile, another stream consisting of compound keys and their positions is created and used to generate models with Algorithm 2 (Line 3). Once the models are yielded by Algorithm 2, they are immediately written to the index file, constituting the bottom layer of the run's learned index. Then, we recursively build the upper layers of the index until the top layer can fit into a single disk page (Lines 4 to 8). Specifically, for each layer, we scan lower-layer models (denoted as $\mathcal{F}_I[-n:]$) to create a compound key stream using k_{min} in each model and their index file positions (Line 6). Similar to the bottom layer, we use Algorithm 2 on the stream to create models and instantly write them to the index file (Line 7). This results in the sequential storage of models across layers in a bottom-up manner. The index file remains valid from its construction until the next level merge operation thanks to the LSM-tree-based maintenance approach, which avoids costly model retraining.

4.2 Merkle File Construction

A Merkle file stores an m-ary complete MHT that authenticates the compound key-value pairs in the corresponding value file. The related index file's learned models are excluded from authentication, as they solely enhance query efficiency and do not affect blockchain data integrity. For the m-ary complete MHT, the bottom layer consists of hash values of every compound key-value pair in the value file. The hash values in an upper layer are recursively computed from every m hash values in the lower layer, except that the last one might be computed from less than m hash values in the lower layer.

Definition 2 (Hash Value)**.** *A hash value in the bottom layer of the MHT is computed as $h_i = h(\mathcal{K}_i \| value_i)$, where $\mathcal{K}_i, value_i$ are the corresponding compound key and value, $\|$ is the concatenation operator, and $h(\cdot)$ is a cryptographic hash function such as SHA-256. A hash value in an upper layer of the MHT is computed as $h_i = h(h_i^1 \| h_i^2 \| \cdots \| h_i^{m^*})$, where $m^* \leq m$ and h_i^j is the corresponding j-th hash in the lower layer.*

Algorithm 4: Merkle File Construction

```
1  Function ConstructMerkleFile(S, n, m)
       Input: Input stream S of compound key-value pairs,
              stream size n, fanout m
       Output: Merkle file F_H
2      N_nodes ← [n, ⌈n/m⌉, ⌈n/m^2⌉, ···, 1], d ← |N_nodes|;
3      layer_offset[0] ← 0;
4      layer_offset[i] ← Σ_0^{i−1} N_nodes[i−1], ∀i ∈ [1, d−1];
5      Create a merkle file F_H with size Σ_{i=0}^{d−1} N_nodes[i];
6      Create a cache C with d number of buffers;
7      foreach ⟨K, value⟩ ← S do
8          h' ← h(K||value), append h' to C[0];
9          foreach i in 0 to d−2 do
10             if |C[i]| = m then
11                 h' ← h(C[i]), append h' to C[i+1];
12                 Flush C[i] to F_H at offset layer_offset[i];
13                 layer_offset[i] ← layer_offset[i] + m;
14             else break;
15     foreach i in 0 to d−1 do
16         if C[i] is not empty then
17             h' ← h(C[i]), append h' to C[i+1];
18             Flush C[i] to F_H at offset layer_offset[i];
19     return F_H;
```

Similar to Algorithm 3, we streamingly generate the Merkle file. However, instead of layer-wise construction, we concurrently build all MHT layers to reduce IO costs, as shown in Algorithm 4. Note that the size of the input stream of compound key-value pairs n is known in advance since the size of a value file is determined by the level of its corresponding run. Thus, the MHT has $\lceil \log_m n \rceil + 1$ layers, containing $n, \lceil \frac{n}{m} \rceil, \lceil \frac{n}{m^2} \rceil, \cdots, 1$ hash values (Line 2). Layer offsets can also be computed (Lines 3 to 4). For concurrent construction, $\lceil \log_m n \rceil + 1$ buffers are maintained, one per layer. Upon the arrival of a new compound key-value pair, its hash value is computed and added to the bottom layer's buffer (Line 8). When a buffer fills with m hash values, an upper layer's hash value is created and added to its buffer (Line 11). Next, the buffered hash values in the current layer are flushed to the Merkle file, followed by incrementing the offset (Lines 12 to 13). This process recurs in upper layers until a layer with less than m buffered hash values is encountered. Once the input stream is fully processed, any remaining non-empty buffers will hold fewer than m hash values. If so, we'll initiate this process by taking a buffer from the lowest layer and iteratively generating hash values. Each hash value is added to the upper layer before flushing the buffer to the Merkle file (Lines 15 to 18).

Example. *Figure 6 shows an example of a 2-ary MHT with states s_1 to s_4. According to the MHT's structure, $N_{nodes} = [4,2,1]$ and $layer_offset = [0,4,6]$. Assume that s_1, s_2 are already added. In this case, $\mathcal{F}_H$ has h_1, h_2 and cache $C[1]$ con-*

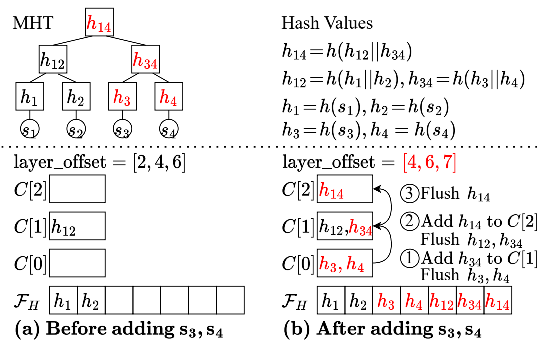


Figure 6: An Example of Merkle File Construction

tains h_{12}, where h_1, h_2 are the hash values of s_1, s_2 and h_{12} is derived from h_1, h_2 (Figure 6(a)). Meanwhile, layer_offset$[0]$ has been updated to 2. After s_3, s_4 are added, their hashes h_3, h_4 will be inserted to cache $C[0]$, resulting in $C[0]$ having 2 hash values. Thus, h_{34} derived from h_3, h_4 will be added into cache $C[1]$ and h_3, h_4 are then flushed to $\mathcal{F}_H$ at offset 2 (step ①). Since $C[1]$ also has 2 hash values so the derived h_{14} is added to cache $C[2]$ and h_{12}, h_{34} are flushed to $\mathcal{F}_H$ at offset layer_offset$[1] = 4$ (step ②). Finally, h_{14} in $C[2]$ is flushed to $\mathcal{F}_H$ at offset layer_offset$[2] = 6$ (step ③).

4.3 Discussions

As discussed earlier, COLE adopts the LSM-tree-based maintenance approach to optimize data writes and disk operations under the column-based design. However, it also comes with some tradeoffs. The presence of multiple levels can impact read performance, as retrieving a state requires traversing multiple levels until a satisfactory result is found. Additionally, the merge operation complicates the process of state rewind, as data cannot be deleted in-place. Therefore, COLE does not support blockchain forking and is designed to work with blockchains that do not fork [5, 22, 53].

We next discuss the ACID properties in COLE. COLE achieves atomicity by maintaining root_hash_list in an atomic manner. During the level merge process, root_hash_list is updated atomically only after constructing all three files in the new level, followed by removing the old level files. This ensures data consistency as the old level files remain intact and are referenced by root_hash_list even during a node crash. Concurrency control is not required due to the write-serializability guarantee of the consensus protocol. Data integrity is ensured using Merkle-based structures for each level. For durability, COLE uses transaction logs as the Write Ahead Log since they are agreed upon by the consensus protocol. In case of a crash, COLE recovers by replaying transactions since the last checkpoint. A checkpoint is created when the in-memory MB-tree is flushed to the first disk level and cleared. At this time point, all the data in the system is safely stored on the disk. After a crash, COLE reverts to the last checkpoint, discards all the files in the unfinished merge levels, and starts fresh with an empty in-memory MB-tree. It then replays all unprocessed transactions and restarts the aborted level merges.

5 Write with Asynchronous Merge

Algorithm 1 may trigger recursive merge operations during some writes (e.g., steps ① and ② in Figure 4). As a result, it can introduce long-tail latency and cause all future operations to stall. This issue is known as *write stall*, which leads to periodic drops in application throughput to near-zero levels and dramatic fluctuations in system performance. A common solution is to make the merge operations asynchronous by moving them to separate threads. However, the existing asynchronous merge solution is not suitable for blockchain applications. Since different nodes in the blockchain network could have drastically different computation capabilities, the storage structure will become out-of-sync among nodes when applying asynchronous merges. This will result in different H_{state}'s and break the requirement of the blockchain protocol.

To address these challenges, we design a novel asynchronous merge algorithm for COLE, which ensures the synchronization of the storage across blockchain nodes. The algorithm introduces two checkpoints, *start* and *commit*, within the asynchronous merge process for each on-disk level. By synchronizing the checkpoints, we ensure consistent blockchain storage and thus H_{state} agreed by the network. To further minimize the possibility of long-tail latency due to delays at the commit checkpoint, we propose to make the interval between the start checkpoint and the commit checkpoint proportional to the size of the run. This ensures that the majority of the nodes in the network can complete the merge operation before reaching the commit checkpoint.

To realize our idea, we propose to have each level of COLE contain two groups of runs as shown in Figure 7. Each group's design is identical to the one discussed in Section 4. Specifically, the in-memory level now contains two groups of MB-tree, each with a capacity of B states. Similarly, each on-disk level contains two groups of up to T sorted runs. Level i can hold a maximum of $2 \cdot B \cdot T^i$ states. The two groups in each level have two mutually exclusive roles, namely *writing* and *merging*. The writing group accepts newly created runs from the upper level. On the other hand, the merging group generates a new run from its own data and adds to the writing group of the next level in an asynchronous fashion.

Algorithm 5 shows the write operation in COLE with asynchronous merge. First, new state values are inserted into the current writing group of in-memory level L_0 (Lines 2 to 4). The levels in COLE are then traversed from smaller to larger. When a level is full, we commit the previous merge operation in the current level and start a new merge operation in a new thread. To accommodate slow nodes in the network, we check if the previous merging thread of the current level exists and is still in progress, and wait for it to finish if necessary (Line 9).

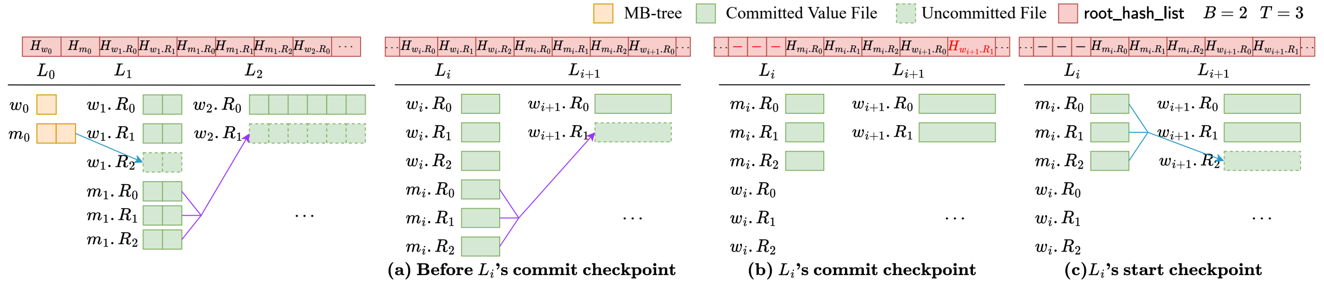


Figure 7: Asynchronous Merge

Figure 8: An Example of Asynchronous Merge

Algorithm 5: Write Algorithm with Asynchronous Merge

1 **Function** Put (*addr*, *value*)
  Input: State address *addr*, value *value*
2   $blk \leftarrow$ current block height; $\mathcal{K} \leftarrow \langle addr, blk \rangle$;
3   $w_0 \leftarrow$ Get L_0's writing group;
4   Insert $\langle \mathcal{K}, value \rangle$ into the MB-tree of w_0;
5   $i \leftarrow 0$;
6   **while** w_i *becomes full* **do**
7     $m_i \leftarrow$ Get L_i's merging group;
8   **if** m_i*.merge_thread exists* **then**
9   Wait for m_i.merge_thread to finish;
10   Add the root hash of the generated run from m_i.merge_thread to root_hash_list;
11   Remove the root hashes of the runs in m_i from root_hash_list;
12   Remove all the runs in m_i;
13   Switch m_i and w_i;
14   m_i.merge_thread $\leftarrow$ **start thread do**
15   **if** $i = 0$ **then**
16   Flush the leaf nodes in m_i to L_{i+1}'s writing group a sorted run;
17   Generate files $\mathcal{F}_V, \mathcal{F}_I, \mathcal{F}_H$ for the new run;
18   **else**
19   Sort-merge all the runs in m_i to L_{i+1}'s writing group a new run;
20   Generate files $\mathcal{F}_V, \mathcal{F}_I, \mathcal{F}_H$ for the new run;
21     $i \leftarrow i + 1$;
22   Update H_{state} when finalizing the current block;

The previous merge operation is committed by adding the root hash of the newly generated run to root_hash_list (Line 10), while obsolete run hashes are removed from root_hash_list (Line 11) and the obsolete runs in the merging group are also removed (Line 12). The above procedure ensures the commit checkpoint occurs simultaneously across nodes in the network, which is essential to synchronize the blockchain states and the corresponding root digest. Following this, the roles between the two groups in the current level are switched (Line 13). This means that future write operations will be directed to the vacated space of the new writing group, whereas the merge operation will be performed on the new merging group, which is now full. The latter starts a new merge thread, whose procedure is similar to that of Algorithm 1 (Lines 14 to 20). Lastly, when finalizing the current block, H_{state} is updated using stored root hashes in root_hash_list (Line 22).

Example. *Figure 8 shows an example of the asynchronous merge from level L_i to L_{i+1}, where $T = 3$. The uncommitted files are denoted by dashed boxes. Figure 8(a) shows COLE's structure before L_i's commit checkpoint, when L_i's writing group w_i becomes full. In case m_i's merging thread (denoted by the purple arrow) is not yet finished, we wait for it to finish. Then, during L_i's commit checkpoint, $w_{i+1}.R_1$'s root hash is added to root_hash_list and all runs in m_i (i.e., $m_i.R_0, m_i.R_1, m_i.R_2$) are removed (Figure 8(b)). Next, m_i and w_i's roles are switched. Finally, a new thread will be started (denoted by the blue arrow) to merge all runs in m_i to L_{i+1}'s writing group as the third run $w_{i+1}.R_2$ (Figure 8(c)).*

Soundness Analysis. Next, we show our proposed asynchronous merge operation is sound. Specifically, the following two requirements are satisfied.

- The blockchain states' root digest H_{state} is always synchronized among nodes in the blockchain network regardless of how long the underlying merge operation takes.
- The interval between the start checkpoint and the commit checkpoint for each level is proportional to the size of the runs to be merged.

The first requirement ensures blockchain states are solely determined by the current committed states and are independent of individual node performance variations. The second requirement minimizes the likelihood of nodes waiting for merge operations of longer runs. We now prove that our algorithm complies with the requirements.

Proof Sketch. It is trivial to show that the first requirement is satisfied as the update of root_hash_list (hence H_{state}) occurs outside the asynchronous merge thread, making the update of H_{state} fully synchronous and deterministic. For the second requirement, the interval between the start checkpoint and the commit checkpoint in any level equals the time taken to fill up the writing group in the same level. Since the latter contains those runs to be merged in this level, the interval is proportional to the size of the runs. □

6 Read Operations of COLE

In this section, we discuss the read operations of COLE, including the get query and the provenance query with its veri-

Algorithm 6: Get Query

```
1  Function Get (addr)
     Input: State address addr
     Output: State latest value value
2    K_q ← ⟨addr, max_int⟩;
3    foreach g in {L_0's writing group, L_0's merging group} do
4      ⟨K', state'⟩ ← SearchMBTree(g, K_q);
5      if K'.addr = addr then return state';
6    foreach level i in {1, 2, ...} do
7      RS ← {R_{i,j} | R_{i,j} ∈ L_i's writing group ∧ committed};
8      RS ← RS + {R_{i,j} | R_{i,j} ∈ L_i's merging group};
9      foreach R_{i,j} in RS do
10       ⟨⟨K', state'⟩, pos'⟩ ← SearchRun(R_{i,j}, K_q);
11       if K'.addr = addr then return state';
12   return nil;
```

fication function. We assume that COLE is implemented with the asynchronous merge.

6.1 Get Query

Algorithm 6 shows the get query process. As mentioned in Section 3.2, getting a state's latest value requires a special compound key $\mathcal{K}_q = \langle addr_q, max_int \rangle$. Owing to the temporal order of COLE's levels, we perform the search from smaller levels to larger levels, until a satisfied state value is found. This involves searching both the writing and merging groups' MB-trees in the in-memory level L_0 as both of them are committed (Lines 3 to 5). Then, in each on-disk level, a search is performed in the committed writing group's runs, followed by the merging group's runs (Lines 6 to 11). Note that the runs in the same group will be searched in the order of their freshness. For the example in Figure 7, we search the MB-trees in w_0 and m_0, followed by the runs in the order of $w_1.R_1$, $w_1.R_0$, $m_1.R_2$, $m_1.R_1$, $m_1.R_0$, $w_2.R_0$, $\cdots$, while the uncommitted $w_1.R_2, w_2.R_1$ are skipped. The search halts once the satisfied state is found.

To search an on-disk run, we use Algorithm 7. First, if the queried address $addr_q$ is not in the run's bloom filter $\mathcal{B}$, the run is skipped (Line 2). Otherwise, models in the index file $\mathcal{F}_I$ are used to find $\mathcal{K}_q$. The search starts from the top layer of models, stored on the last page of $\mathcal{F}_I$. The model covering $\mathcal{K}_q$ is found by binary searching k_{min} of each model in this page (Line 4). Then, a recursive query on models in subsequent layers is conducted from top to bottom (Lines 5 to 7). Upon reaching the bottom layer, the corresponding model is used to locate the state value in the value file $\mathcal{F}_V$ (Line 8).

Function `QueryModel`$(\cdot)$ in Algorithm 7 shows the procedure of using a learned model $\mathcal{M}$ to locate the queried compound key $\mathcal{K}_q$. If the model covers $\mathcal{K}_q$, it predicts the position pos_{pred} of the queried data (Line 12). With the error bound of the model 2ε equaling the page size, the predicted page id is computed as $pos_{pred}/2\varepsilon$ (Line 13). The corresponding page $\mathcal{P}$ is fetched and the first and last models are checked whether they cover $\mathcal{K}_q$. If not, the adjacent page is fetched as

Algorithm 7: Search a Run

```
1  Function SearchRun (F_I, F_V, B, K_q)
     Input: Index file F_I, value file F_V, bloom filter B,
            compound key K_q = ⟨addr_q, blk_q⟩
     Output: Queried state s and its position pos
2    if addr_q ∉ B then return;
3    K_q ← BigNum(K_q);
4    P ← F_I's last page; M ← BinarySearch(P, K_q);
5    ⟨M, pos⟩ ← QueryModel(M, F_I, K_q);
6    while pos is not pointing to the bottom models do
7      ⟨M, pos⟩ ← QueryModel(M, F_I, K_q);
8    return QueryModel(M, F_V, K_q);
9  Function QueryModel (M, F, K_q)
     Input: Model M, query file F, compound key K_q
     Output: Queried data and its position in F
10   ⟨sl, ic, k_min, p_max⟩ ← M;
11   if K_q < k_min then return;
12   pos_pred ← min(K_q · sl + ic, p_max);
13   page_pred ← pos_pred / 2ε;
14   P ← F's page at page_pred;
15   if K_q < P[0].k then
16     P ← F's page at page_pred − 1;
17   else if K_q > P[−1].k then
18     P ← F's page at page_pred + 1;
19   return BinarySearch(P, K_q);
```

$\mathcal{P}$ (Lines 15 to 18). This process involves at most two pages for prediction, hence minimizing IO. Finally, a binary search in $\mathcal{P}$ locates the queried data (Line 19).

6.2 Provenance Query

A provenance query resembles a get query but with notable distinctions. Unlike a get query, a provenance query involves a range search based on the queried block height range. This entails computing two boundary compound keys, $\mathcal{K}_l = \langle addr, blk_l - 1 \rangle$ and $\mathcal{K}_u = \langle addr, blk_u + 1 \rangle$, with offsets adjusted by one to prevent the omission of valid results. Moreover, a provenance query provides Merkle proofs to authenticate the results.

Specifically, during the search of MB-trees in L_0, in addition to retrieving satisfactory results, Merkle paths are included in the proof using a similar approach mentioned in Section 2. For the runs of the on-disk levels, we search in the same order as those described in Algorithm 6. $\mathcal{K}_l$ is used as the search key when applying the learned models to find the first query result in each run. Then, the value file is scanned sequentially until a state beyond $\mathcal{K}_u$ is reached.[1] Afterwards, a Merkle proof is computed upon the first and last results' positions pos_l, pos_u of each run. Since the states in the value file and their hash values in the Merkle file share the same position, the Merkle paths of the hash values at pos_l and pos_u are used as the Merkle proof. To compute the Merkle path,

[1] For simplicity, we assume that *addr* is in the bloom filter $\mathcal{B}$. If not, $\mathcal{B}$ is also added as the proof to prove that *addr* is not in the run.

Cost	MPT	COLE	COLE w/ async-merge
Storage size	$O(n \cdot d_{MPT})$	$O(n)$	
Write IO cost	$O(d_{MPT})$	$O(d_{COLE})$	
Write tail latency	$O(1)$	$O(n)$	$O(1)$
Write memory footprint	$O(1)$	$O(T + m \cdot d_{COLE})$	$O(T \cdot d_{COLE} + m \cdot d_{COLE}^2)$
Get query IO cost	$O(d_{MPT})$	$O(T \cdot d_{COLE} \cdot C_{model})$	
Prov-query IO cost	$O(d_{MPT})$	$O(T \cdot d_{COLE} \cdot C_{model} + m \cdot d_{COLE}^2)$	
Prov-query proof size	$O(d_{MPT})$	$O(m \cdot d_{COLE}^2)$	

Table 1: Complexity Comparison

we traverse the MHT in the Merkle file from bottom to top. Note that given a hash value's position *pos* at layer *i*, we can directly compute its parent hash value's position in the Merkle file as $\lfloor (pos - \sum_0^{i-1} \lceil \frac{n}{m^i} \rceil)/m \rfloor + \sum_0^i \lceil \frac{n}{m^i} \rceil$. Due to the space limitation, the detailed procedure of the provenance query is given in our technical report [63] .

On the user's side, the verification algorithm works as follows: (1) use each MB-tree's results and their corresponding Merkle proof to reconstruct the MB-tree's root hash; (2) use each searched run's results and their corresponding Merkle proof to reconstruct the run's root hash; (3) use the reconstructed root hashes to reconstruct the states' root digest and compare it with the published one, H_{state}, in the block header; (4) check the boundary results of each searched run against the compound key range $[\mathcal{K}_l, \mathcal{K}_u]$ to ensure no missing results. If all these checks pass, the results are verified.

7 Complexity Analysis

In this section, we analyze the complexity in terms of storage, memory footprint, and IO cost. To ease the analysis, we assume n as the total historical values, T as the level size ratio, B as the in-memory level's capacity, and m as COLE's MHT fanout. Table 1 shows the comparison of MPT, COLE, and COLE with the asynchronous merge.

We first analyze the storage size. Since MPT duplicates the nodes of the update path for each insertion, its storage has a size of $O(n \cdot d_{MPT})$, where d_{MPT} is the height of the MPT. COLE completely removes the node duplication, thus achieving an $O(n)$ storage size.

Next, we analyze the write IO cost. MPT takes $O(d_{MPT})$ to write the nodes in the update path, while COLE takes $O(d_{COLE})$ for the worst case when all levels are merged, where d_{COLE} is the number of levels in COLE. Similar to the traditional LSM-tree's write cost [13], the level merge in COLE takes an amortized $O(1)$ IO cost to write the value file, the index file, and the Merkle file. The number of levels d_{COLE} is $\lceil \log_T(\frac{n}{B} \cdot \frac{T-1}{T}) \rceil$, which is logarithmic to n. Note that normally $d_{COLE} < d_{MPT}$ since d_{MPT} depends on the data's key size, which can be large (e.g., when having a 256-bit key, maximum d_{MPT} is 64 under hexadecimal base while COLE has only a few levels following the LSM-tree).

Regarding the write tail latency, MPT has a constant cost since there is no write stall during data writes. On the other hand, COLE may experience the write stall in the worst case, which requires waiting for the merge of all levels and results in the reading and writing of $O(n)$ states. The asynchronous merge algorithm removes the write stall by merging the levels in background threads and reduces the tail latency to $O(1)$.

As for the write memory footprint, MPT has a constant cost since the update nodes are computed on the fly and can be removed from the memory after being flushed to the disk. For COLE, we consider the case of merging the largest level as this is the worst case. The sort-merge takes $O(T)$ memory and the model construction takes constant memory [40]. Constructing the Merkle file takes $O(m \cdot d_{COLE})$ since there are logarithmic layers of cache buffers and each buffer contains m hash values. To sum up, COLE takes $O(T + m \cdot d_{COLE})$ memory during a write operation. For COLE with the asynchronous merge, the worst case is that each level has a merging thread, thus requiring d_{COLE} times of memory compared with the synchronous merge, i.e., $O(T \cdot d_{COLE} + m \cdot d_{COLE}^2)$.

We finally analyze the read operations' costs, including the get query IO cost, the provenance query IO cost, and the proof size of the provenance query. MPT's costs are all linear to the MPT's height, $O(d_{MPT})$. For COLE, T runs in each level should be queried, where we assume that each run takes C_{model} to locate the state. Therefore, the cost of the get query is $O(T \cdot d_{COLE} \cdot C_{model})$. To generate the Merkle proof during the provenance query, an additional $O(m \cdot d_{COLE}^2)$ is required since there are multiple layers of MHT in all levels and $O(m)$ hash values are retrieved for each MHT's layer. The proof size is $O(m \cdot d_{COLE}^2)$ for a similar reason.

8 Evaluation

In this section, we first describe the experiment setup, including comparing baselines, implementation, parameter settings, workloads, and evaluation metrics. Then, we present the experiment results.

8.1 Experiment Setup

8.1.1 Baselines

We compare COLE with the following baselines:

- MPT: It is used by Ethereum to index the blockchain storage. The structure is made persistent as mentioned in Section 1.
- LIPP: It applies LIPP [54], the state-of-the-art learned index supporting *in-place* data writes, to the blockchain storage without our column-based design. LIPP retains the node persistence strategy to support provenance queries.
- Column-based Merkle Index (CMI): It uses the column-based design with traditional Merkle indexes rather than the learned index. It adopts a two-level structure. The upper index is a non-persistent MPT whose key is the state address and the value is the root hash of the lower index. The lower index follows the column-based design, using an MB-tree to store the state's historical values in a

Parameters	**Value**
# of generated blocks	$10^2, 10^3, 10^4, \mathbf{10^5}$
Size ratio T	$2, \mathbf{4}, 6, 8, 10, 12$
COLE's MHT fanout m	$2, \mathbf{4}, 8, 16, 32, 64$

Table 2: System Parameters

contiguous fashion [29].

8.1.2 Implementation and Parameter Setting

We implement COLE and the baselines in Rust programming language. The source code is available at https://github.com/hkbudb/cole. We use the Rust Ethereum Virtual Machine (EVM) to execute transactions, simulating blockchain data updates and reads [2]. Transactions are packed into blocks, each containing 100 transactions. Ten smart contracts are initially deployed and repeatedly invoked with transactions. Big number operations mentioned in Section 3.2 are implemented using the *rug* library [3]. Baselines utilize RocksDB [18] as the underlying storage, while COLE uses simple files for data storage as enabled by our design.

We set $\varepsilon = 23$ based on the page size (4KB) and the compound key-pair size (88 bytes). By default, the size ratio T and the MHT fanout m of COLE are set to 4. Following the default configuration of RocksDB, its memory budget is set to 64MB. The in-memory capacity B is set to the number of states that can fit within the same memory budget. Table 2 shows all the parameters where the default settings are highlighted in bold font. All experiments are run on a machine equipped with an Intel i7-10710U CPU, 16GB RAM, and Samsung SSD 256GB.

8.1.3 Workloads and Evaluation Metrics

The experiment evaluation includes two parts: the overall performance of transaction executions and the performance of provenance queries. For the first part, SmallBank and KVStore from Blockbench [17] are used as macro benchmarks to generate the transaction workload. SmallBank simulates the account transfers while KVStore uses YCSB [9] for read/write tests. YCSB involves a loading phase where base data is generated and stored, followed by a running phase for read/update operations. A transaction that reads/updates data is denoted as a read/write transaction. We set 10^5 transactions as the base data and vary read/update ratios to simulate different scenarios: (i) Read-Write with equal read/write transactions; (ii) Read-Only with only read transactions; and (iii) Write-Only with all write transactions. The overall performance is evaluated in terms of the average transaction throughput, the tail latency, and the storage size.

To evaluate provenance queries, we use KVStore to simulate the workload including frequent data updates. We initially write 100 states as the base data and then continuously generate write transactions to update the base data's states. For each query, we randomly select a key from the base data and vary the block height range (e.g., $2, 4, \cdots, 128$), which follows [44]'s setting. The evaluation metrics include (i) CPU time of the query executed on the blockchain node and verified by the query user and (ii) proof size.

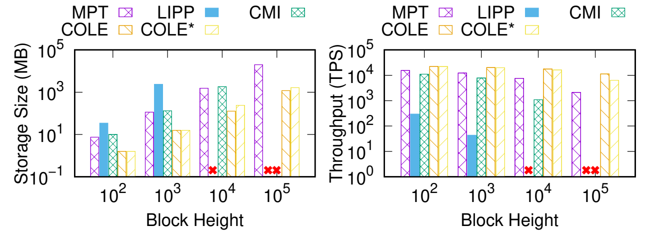


Figure 9: Performance vs. Block Height (SmallBank)

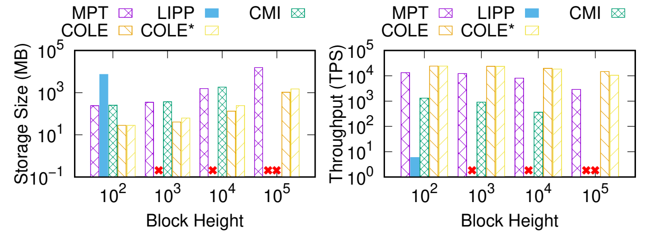


Figure 10: Performance vs. Block Height (KVStore)

8.2 Experimental Results

8.2.1 Overall Performance

Figures 9 and 10 show the storage size and throughput of COLE and all baselines under the SmallBank and KVStore workloads, respectively. We denote COLE with the asynchronous merge as COLE*.

We make several interesting observations. First, COLE significantly reduces the storage size compared to MPT as the blockchain grows. For example, at a block height of 10^5, the storage size decreases by 94% and 93% for SmallBank and KVStore, respectively. This is due to COLE's elimination of the need to persist internal data structures via the column-based design, and its use of storage-efficient learned models for indexing. Moreover, COLE outperforms MPT in throughput, achieving a $1.4\times$-$5.4\times$ improvement, thanks to its learned index. COLE* performs slightly worse than COLE due to the overhead of the asynchronous merge.

Second, using the learned index without the column-based design (LIPP) even increases the blockchain storage. At a block height of 10^2, the storage size of LIPP already exceeds MPT's by $5\times$ (for SmallBank) and $31\times$ (for KVStore). This happens because the learned index often generates larger index nodes that must be persisted with each new block, leading to increased storage and significant IO operations. Consequently, LIPP's throughput is significantly worse than MPT. We are not able to report the results of LIPP for the block height above 10^3 for SmallBank and 10^2 for KVStore as the experiment could not be finished within 24 hours.

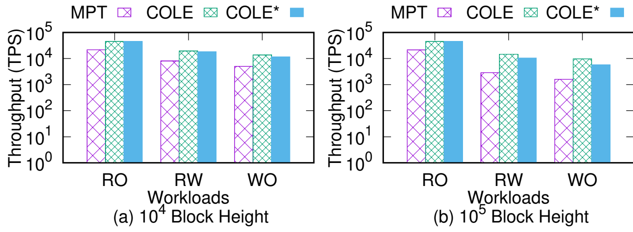


Figure 11: Throughput vs. Workloads (KVStore)

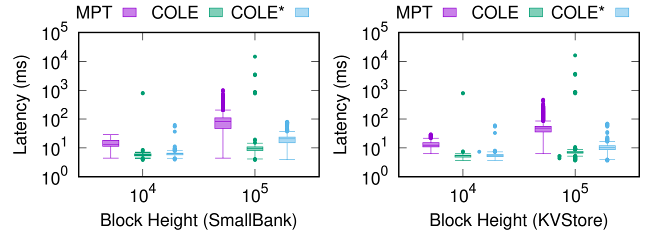


Figure 12: Latency Box Plot

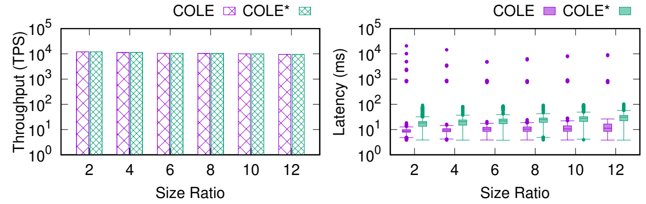


Figure 13: Impact of Size Ratio

Third, extending MPT with the column-based design (CMI) does not significantly change the storage size. The additional storage of the lower-level MB-tree and the use of the RocksDB backend largely negate the benefit of removing node persistence. Additionally, refreshing Merkle hashes of all nodes in the index update path, which entails both read and write IOs, further impacts performance. Consequently, the throughput of CMI is 7× and 22× worse than MPT for SmallBank and KVStore, respectively, at a block height of 10^4. The experiments of CMI cannot scale beyond a block height of 10^4.

Overall, with a unique combination of the learned index, column-based design, and write-optimized strategies, COLE and COLE* not only achieve the smallest storage requirement but also gain the highest system throughput.

8.2.2 Impact of Workloads

We use KVStore to evaluate the impact of different workloads, namely Read-Only (RO), Read-Write (RW), and Write-Only (OW), in terms of the system throughput. As shown in Figure 11, the throughputs of all systems decrease with more write operations in the workload. The performance of MPT degrades by up to 93% while that of COLE and COLE* degrades by up to 87%. This shows that the LSM-tree-based maintenance approach helps optimize the write operation. We omit LIPP and CMI in Figure 11 since they cannot scale beyond a block height of 10^3 and 10^4, respectively.

8.2.3 Tail Latency

To assess the effect of the asynchronous merge, Figure 12 shows the box plot of the latency of SmallBank and KVStore workloads at block heights of 10^4 and 10^5. The tail latency is depicted as the maximum outlier. As the blockchain grows, COLE* decreases the tail latency by 1-2 orders of magnitude for both workloads. This shows that the asynchronous merge strategy will become more effective when the system scales up for real-world applications. Owing to the asynchronous merge overhead, COLE* incurs slightly higher median latency than COLE, but it still outperforms MPT.

8.2.4 Impact of Size Ratio

Figure 13 shows the system throughput and latency box plot under 10^5 block height using the SmallBank benchmark with varying size ratio T. As the size ratio increases, the throughput remains stable, while the tail latency shows a U shape. We observe that $T = 6$ and $T = 4$ are the best settings for COLE and COLE*, respectively, with the lowest tail latency. Meanwhile, with an increasing size ratio, the median latency of both COLE and COLE* increases.

8.2.5 Provenance Query Performance

We now evaluate the performance of provenance queries by querying historical state values of a random address within the latest q blocks. With the current block height fixed at 10^5, we vary q from 2 to 128. LIPP and CMI are omitted here since they cannot scale at 10^5 block height. Figure 14 shows that MPT's CPU time and proof size grow linearly with q while those of COLE and COLE* grow only sublinearly. This is because MPT requires to query each block inside the queried range. In contrast, COLE and COLE*'s column-based design often locates query results within contiguous storage of each run, hence reducing the number of index traversals during the query and shrinking the proof size by sharing ancestor nodes in the Merkle path. COLE and COLE*'s proof sizes surpass that of MPT when the query range is small due to limited sharing capabilities within a small query range.

9 Related Work

In this section, we briefly review the related works on learned indexes and blockchain storage management.

9.1 Learned Index

Learned index has been extensively studied in recent years. The original learned index [26] only supports static data while PGM-index [20], Fiting-tree [21], ALEX [15], LIPP [54], and LIFOSS [61] support dynamic data using different strategies. All these works are designed and optimized for in-

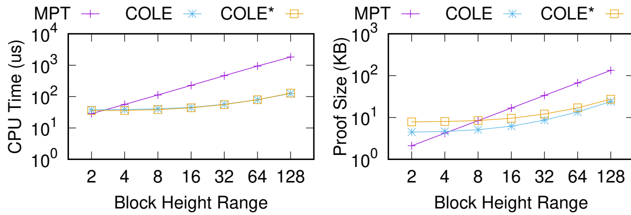


Figure 14: Prov-Query Performance vs. Query Range

memory databases. Bourbon [10] uses the PGM-based models to speed up the lookup in the WiscKey system, which is a persistent key-value store. [27] investigates how existing dynamic learned indexes perform on-disk and shows the design choices. Some other learned indexes are proposed for more complex application scenarios like spatial data [23, 32, 47], multi-dimensional data [16, 39], and variable-length string data [51]. Moreover, [30, 34] consider designing learned indexes for concurrent systems. [65] proposes a persistent learned index that is specifically designed for the NVM-only architecture with concurrency control. More recently, [31] designs a scalable RDMA-oridented learned key-value store for disaggregated memory systems. Nevertheless, existing works cannot be directly applied to blockchain storage since they do not take into account disk-optimized storage, data integrity, and provenance queries simultaneously.

9.2 Blockchain Storage Management

Pioneering blockchain systems, such as Bitcoin [38] and Ethereum [52], use MPT and store it using simple key-value storage like RocksDB [18], which implements the LSM-tree structure. While many works propose to optimize the generic LSM-tree for high throughput and low latency [12–14, 46, 60], and some propose orthogonal designs that could potentially be incorporated into COLE, they are not specifically designed to meet the unique integrity and provenance requirements of blockchain systems. On the other hand, a large body of research has been carried out to study alternative solutions to reduce blockchain storage overhead. Several studies [11, 19, 24, 25, 62] consider using *sharding* techniques to horizontally partition the blockchain storage and each partition is maintained by a subset of nodes, thus reducing the overall storage overhead. Distributed data storage [43, 59] or moving on-chain states to off-chain nodes [6, 8, 48, 56, 58] has also been proposed to reduce each blockchain node's storage overhead. Besides, ForkBase [50] proposes to optimize blockchain storage by deduplicating multi-versioned data and supporting efficient fork operations. [28] employs a vector commitment protocol and multi-level authenticated trees to reduce I/O costs for blockchain storage. To the best of our knowledge, COLE is the first work that targets the index itself to address the blockchain storage overhead.

Another related topic is to support efficient queries in blockchain systems. LineageChain [44] focuses on provenance queries in the blockchain. Verifiable boolean range queries are studied in vChain and vChain+ [49, 55], where accumulator-based authenticated data structures are designed. GEM^2-tree [64] explores query processing in the context of on-chain/off-chain hybrid storage. FalconDB [42] combines the blockchain and the collaborative database to support SQL queries with a strong security guarantee. [57] studies the authenticated spatial and keyword queries in blockchain databases. iQuery [35] supports intelligent blockchain analytical queries and guarantees the trustworthiness of query results by using multiple service providers. While all these works focus on proposing additional data structures to process specific queries, COLE focuses on improving the performance of the general blockchain storage system.

10 Conclusion

In this paper, we have designed COLE, a novel column-based learned storage for blockchain systems. COLE follows the column-based database design to contiguously store each state's historical values using an LSM-tree approach. Within each run of the LSM-tree, a disk-optimized learned index has been designed to facilitate efficient data retrieval and provenance queries. Moreover, a streaming algorithm has been proposed to construct Merkle files that are used to ensure blockchain data integrity. In addition, a new checkpoint-based asynchronous merge strategy has been proposed to tackle the long-tail latency issue for data writes in COLE. Extensive experiments show that, compared with the existing systems, the proposed COLE system reduces the storage size by up to 94% and improves the system throughput by 1.4×-5.4×. Additionally, the proposed asynchronous merge decreases the long-tail latency by 1-2 orders of magnitude while maintaining a comparable storage size.

For future work, we plan to extend COLE to support blockchain systems that undergo forking, where the states of a forked block can be rewound. We will investigate efficient strategies to remove the rewound states from storage. Furthermore, since the column-based design stores blockchain states contiguously, compression techniques can be applied to take advantage of similarities between adjacent data. We will study how to incorporate compression strategies into the learned index.

Acknowledgments

This work is supported by Hong Kong RGC Grants (Project No. 12200022, 12201520, C2004-21GF). Jianliang Xu is the corresponding author.

References

[1] Ethereum full node sync (archive) chart. https://etherscan.io/chartsync/chainarchive, 2023.

[2] Ethereum virtual machine. https://github.com/rust-blockchain/evm, 2023.

[3] rug library. https://docs.rs/rug, 2023.

[4] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, pages 1664–1665, 2009.

[5] 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*, pages 1–15, 2018.

[6] 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, 2019.

[7] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, pages 398–461, 2002.

[8] Alexander Chepurnoy, Charalampos Papamanthou, Shravan Srinivasan, and Yupeng Zhang. Edrax: A cryptocurrency with stateless transaction validation. *Cryptology ePrint Archive*, 2018.

[9] 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*, pages 143–154, 2010.

[10] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. From WiscKey to bourbon: A learned index for Log-Structured merge trees. In *OSDI*, pages 155–171, 2020.

[11] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*, pages 123–140, 2019.

[12] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *ACM SIGMOD*, pages 79–94, New York, NY, USA, 2017.

[13] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *ACM SIGMOD*, pages 505–520, 2018.

[14] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. Spooky: granulating lsm-tree compactions correctly. *PVLDB*, pages 3071–3084, 2022.

[15] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. ALEX: an updatable adaptive learned index. In *ACM SIGMOD*, pages 969–984, 2020.

[16] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *PVLDB*, page 74–86, 2020.

[17] 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 *ACM SIGMOD*, pages 1085–1100, 2017.

[18] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, pages 1–32, 2021.

[19] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. BlockchainDB: A shared database on blockchains. *PVLDB*, pages 1597–1609, 2019.

[20] Paolo Ferragina and Giorgio Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB*, pages 1162–1175, 2020.

[21] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Fiting-tree: A data-aware index structure. In *ACM SIGMOD*, pages 1189–1206, 2019.

[22] 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*, pages 51–68, 2017.

[23] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. The rlr-tree: A reinforcement learning based r-tree for spatial data. In *ACM SIGMOD*, 2023.

[24] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. ResilientDB: Global scale resilient blockchain fabric. *PVLDB*, page 868–883, 2020.

[25] Zicong Hong, Song Guo, Enyuan Zhou, Wuhui Chen, Huawei Huang, and Albert Zomaya. Gridb: Scaling blockchain database via sharding and off-chain cross-shard mechanism. *PVLDB*, pages 1685–1698, 2023.

[26] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *ACM SIGMOD*, pages 489–504, 2018.

[27] Hai Lan, Zhifeng Bao, J Shane Culpepper, and Renata Borovica-Gajic. Updatable learned indexes meet disk-resident dbms-from evaluations to design choices. *Proceedings of the ACM on Management of Data*, pages 1–22, 2023.

[28] Chenxing Li, Sidi Mohamed Beillahi, Guang Yang, Ming Wu, Wei Xu, and Fan Long. {LVMT}: An efficient authenticated storage for blockchain. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 135–153, 2023.

[29] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *ACM SIGMOD*, pages 121–132, 2006.

[30] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. FINEdex: a fine-grained learned index scheme for scalable and concurrent memory systems. *PVLDB*, pages 321–334, 2021.

[31] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. ROLEX: A scalable RDMA-oriented learned Key-Value store for disaggregated memory systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 99–114, 2023.

[32] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. LISA: A learned index structure for spatial data. In *ACM SIGMOD*, pages 2119–2133, 2020.

[33] Yinan Li, Bingsheng He, Qiong Luo, and Ke Yi. Tree indexing on flash disks. In *IEEE ICDE*, pages 1303–1306, 2009.

[34] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. Apex: a high-performance learned index on persistent memory. *PVLDB*, pages 597–610, 2021.

[35] Lingling Lu, Zhenyu Wen, Ye Yuan, Binru Dai, Peng Qian, Changting Lin, Qinming He, Zhenguang Liu, Jianhai Chen, and Rajiv Ranjan. Iquery: A trustworthy and scalable blockchain analytics platform. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[36] Raghav Mehra, Nirmal Lodhi, and Ram Babu. Column based nosql database, scope and future. *International Journal of Research and Analytical Reviews*, pages 105–113, 2015.

[37] Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238, 1989.

[38] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[39] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning multi-dimensional indexes. In *ACM SIGMOD*, pages 985–1000, 2020.

[40] Joseph O'Rourke. An on-line algorithm for fitting straight lines between data ranges. *Communications of the ACM*, pages 574–578, 1981.

[41] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, pages 351–385, 1996.

[42] Yanqing Peng, Min Du, Feifei Li, Raymond Cheng, and Dawn Song. Falcondb: Blockchain-based collaborative database. In *ACM SIGMOD*, pages 637–652, 2020.

[43] Xiaodong Qi, Zhao Zhang, Cheqing Jin, and Aoying Zhou. Bft-store: Storage partition for permissioned blockchain via erasure coding. In *IEEE ICDE*, pages 1926–1929, 2020.

[44] Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. Fine-grained, secure and efficient data provenance on blockchain systems. *PVLDB*, pages 975–988, 2019.

[45] Fahad Saleh. Blockchain without waste: Proof-of-stake. *SSRN Electronic Journal*, 2018.

[46] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. Constructing and analyzing the lsm compaction design space. *PVLDB*, pages 2216–2229, 2021.

[47] Yufan Sheng, Xin Cao, Yixiang Fang, Kaiqi Zhao, Jianzhong Qi, Gao Cong, and Wenjie Zhang. Wisk: A workload-aware learned index for spatial keyword queries. *ACM SIGMOD*, pages 1–27, 2023.

[48] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In *International Conference on Security and Cryptography for Networks*, pages 45–64, 2020.

[49] Haixin Wang, Cheng Xu, Ce Zhang, Jianliang Xu, Zhe Peng, and Jian Pei. vChain+: Optimizing verifiable blockchain boolean range queries. In *IEEE ICDE*, pages 1927–1940, 2022.

[50] Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Beng Chin Ooi, and Pingcheng Ruan. Forkbase: an efficient storage engine for blockchain and forkable applications. *PVLDB*, pages 1137–1150, 2018.

[51] Youyun Wang, Chuzhe Tang, Zhaoguo Wang, and Haibo Chen. SIndex: a scalable learned index for string keys. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 17–24, 2020.

[52] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014.

[53] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework. *White Paper*, pages 2327–4662, 2016.

[54] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. Updatable learned index with precise positions. *PVLDB*, pages 1276–1288, 2021.

[55] Cheng Xu, Ce Zhang, and Jianliang Xu. vChain: Enabling verifiable boolean range queries over blockchain databases. In *ACM SIGMOD*, pages 141–158, 2019.

[56] Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. SlimChain: scaling blockchain transactions through off-chain storage and parallel processing. *PVLDB*, pages 2314–2326, 2021.

[57] Hao Xu, Bin Xiao, Xiulong Liu, Li Wang, Shan Jiang, Weilian Xue, Jianrong Wang, and Keqiu Li. Empowering authenticated and efficient queries for stk transaction-based blockchains. *IEEE Transactions on Computers*, 2023.

[58] Zihuan Xu and Lei Chen. L2chain: Towards high-performance, confidential and secure layer-2 blockchain solution for decentralized applications. *PVLDB*, pages 986–999, 2022.

[59] Zihuan Xu, Siyuan Han, and Lei Chen. Cub, a consensus unit-based storage scheme for blockchain system. In *IEEE ICDE*, pages 173–184, 2018.

[60] Jinghuan Yu, Sam H Noh, Young-ri Choi, and Chun Jason Xue. ADOC: Automatically harmonizing dataflow between components in Log-StructuredKey-Value stores for improved performance. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 65–80, 2023.

[61] Tong Yu, Guanfeng Liu, An Liu, Zhixu Li, and Lei Zhao. LIFOSS: a learned index scheme for streaming scenarios. *World Wide Web*, pages 1–18, 2022.

[62] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *ACM CCS*, pages 931–948, 2018.

[63] Ce Zhang, Cheng Xu, Haibo Hu, and Jianliang Xu. COLE: A column-based learned storage for blockchain systems (technical report). https://www.comp.hkbu.edu.hk/~db/cole_tech_report.pdf, 2024.

[64] Ce Zhang, Cheng Xu, Jianliang Xu, Yuzhe Tang, and Byron Choi. GEM^2-Tree: A gas-efficient structure for authenticated range queries in blockchain. In *IEEE ICDE*, pages 842–853, 2019.

[65] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, et al. PLIN: a persistent learned index for non-volatile memory with high performance and instant recovery. *PVLDB*, pages 243–255, 2022.

A Artifact Appendix

Abstract

This artifact includes the source code of the proposed system COLE, along with the source code of other baseline systems used for comparison. Additionally, the artifact provides the procedure for generating the dataset under the YCSB benchmark, which is used for evaluating the performance.

Scope

The artifact is an academic proof-of-concept prototype and has not undergone thorough code review. It should be noted that the implementation is not suitable for production use.

Contents

The artifact consists of the following essential directories:

- *cole-index*: This directory contains the implementation of COLE.
- *cole-star*: This directory corresponds to the implementation of the asynchronous version of COLE.
- *patricia-trie*: The MPT implementation can be found in this directory.
- *lipp*: The implementation of LIPP with node persistence is located in this directory.
- *non-learn-cmi*: This directory contains the implementation of CMI, as mentioned in Section 8.
- *exp*: The evaluation backend for all systems is included in this directory.

Hosting

The artifact is hosted on a GitHub repository with the master branch and the latest commit version.

Requirements

The artifact has been evaluated on Ubuntu 20.04 LTS. Please keep in mind that the scripts provided in the README file for installing dependencies may differ for other platforms.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Baleen: ML Admission & Prefetching for Flash Caches

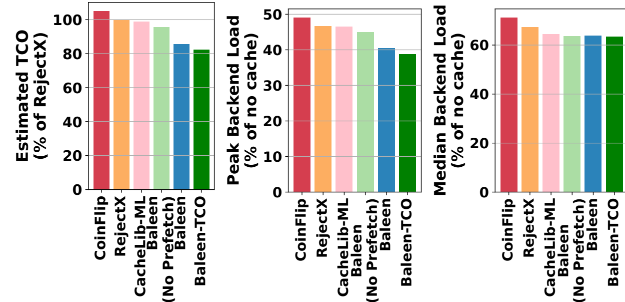
Daniel Lin-Kit Wong[*], *Hao Wu*[†], *Carson Molder*[§], *Sathya Gunasekar*[†], *Jimmy Lu*[†], *Snehal Khandkar*[†]
Abhinav Sharma[†], *Daniel S. Berger*[‡], *Nathan Beckmann, Gregory R. Ganger*
Carnegie Mellon University; [†]Meta; [‡]Microsoft & University of Washington; [§]UT Austin

Abstract

Flash caches are used to reduce peak backend load for throughput-constrained data center services, reducing the total number of backend servers required. Bulk storage systems are a large-scale example, backed by high-capacity but low-throughput hard disks, and using flash caches to provide a more cost-effective storage layer underlying everything from blobstores to data warehouses.

However, flash caches must address the limited write endurance of flash by limiting the long-term average flash write rate to avoid premature wearout. To do so, most flash caches must use admission policies to filter cache insertions and maximize the workload-reduction value of each flash write.

The Baleen flash cache uses coordinated ML admission and prefetching to reduce peak backend load. After learning painful lessons with our early ML policy attempts, we exploit a new cache residency model (which we call *episodes*) to guide model training. We focus on optimizing for an end-to-end system metric (*Disk-head Time*) that measures backend load more accurately than IO miss rate or byte miss rate. Evaluation using Meta traces from seven storage clusters shows that Baleen reduces *Peak Disk-head Time* (and hence the number of backend hard disks required) by 12% over state-of-the-art policies for a fixed flash write rate constraint. Baleen-TCO, which chooses an optimal flash write rate, reduces our estimated total cost of ownership (TCO) by 17%. Code and traces are available via https://www.pdl.cmu.edu/CILES/.



(a) Estimated TCO **(b)** Peak load **(c)** Median load

Figure 1: Baleen-TCO reduces (estimated) TCO by 17% and peak load by 16% over the best baseline on 7 Meta traces by choosing the optimal flash write rate. IO and byte miss rates were reduced by 14% and 2% (Suppl A.1). For the default flash write rate, Baleen reduces peak load by 12% over the best baseline.

1 Introduction

Large-scale storage continues to be predominantly done with hard disks (HDDs), which provide much more cost-effective storage than flash. However, HDDs have low throughput, and each can generally only perform about 100 IOs per second (IOPS). Modern storage systems rely heavily on flash caches to absorb a substantial fraction of requests and thereby reduce the number of disks needed to satisfy the IO workload.

Although a functional cache can be realized using traditional approaches, which assume items can be admitted to the cache arbitrarily, it is important to consider the differing natures of HDDs and flash SSDs. In particular, the IOPS and bandwidth of HDDs has not kept up with increases in their capacity, making disk time a key goal of flash caching more than average IO latency. Flash, on the other hand, provides orders of magnitude higher IOPS, but it wears out as it is written. As a result, expected SSD lifetime projections assume relatively low average write rate limits, such as "three drive-writes per day", meaning 3N TB of writes to a N TB SSD each day. Manufacturers offer SSDs with even lower endurance (e.g., 1 drive write per day) with correspondingly lower prices. All of this translates to a need for smart admission policies to decide which items get written into cache [5, 14]. Popular policies have included random admission and history-based policies that reject items without sufficient recent usage.

Machine learning (ML) policies for flash cache admission have been proposed as a solution for avoiding excessive flash writes. However, caching does not easily map to well-trodden problems in computer vision or natural language processing. In particular, a policy's decision is often affected by its past decisions, and can have synergistic or antagonistic effects on other parts of the system. While in theory this can be addressed with end-to-end and reinforcement learning techniques, in practice, such models require large amounts of human capital and computing resources, and do not necessarily outperform a typical well-tuned production system [6, 21, 25, 28, 29, 57].

Making ML policies introspectable is key to their adoption by systems practitioners [55]. While accurate models are desirable, success also hinges on the correct decisions being posed to the models. *How* one uses ML is key: how to generate training examples from traces, how to arrive at optimal decisions for ML to learn from, which subproblems ML should be applied to, and how to optimize end-to-end systems performance without sacrificing introspectability, debuggability, and efficiency. In Baleen, we decompose the flash caching problem into admission, prefetching, and eviction (§3.3). This helped us align policy decisions to well-understood and efficient ML

[*]Correspondence to dlwong@cs.cmu.edu

techniques for supervised learning. We do, however, want to co-design these different components to reap the full benefits. One may depend on the other to be effective, as we found to be true for ML prefetching and ML admission.

This paper explores ML policies for flash caches in bulk storage systems. We introduce a new analytic approach for access pattern analysis, based on a cache residency model we call *episodes* (§3.4), which groups accesses that correspond to an item's cache residency if admitted. Our approach provides a more complete view of end-to-end flash caching policy performance, and enables us to efficiently model policy behavior under multiple constraints. This is especially useful for flash caches given that the resource burden of an admission is dominated by its flash writes, which is the same whether the item is admitted at the start or end of the episode. From our approach, we develop *OPT* (3.5), an episode-based approximation of optimal admission and train ML admission policies to imitate OPT. We benchmark them against OPT and other baseline admission policies on seven recent real-world storage cluster traces collected over 3 years.

Baleen is our resulting ML-guided Flash cache policy. We evaluate it by its savings in *Peak Disk-head Time* (§3.1), a measure of peak backend load, and we find that a combination of ML-guided admission and ML-guided prefetching provides the largest improvement. In deploying ML, we learned that determining the right optimization metric is not an easy task; an earlier version of Baleen improved IO hit ratio but had worse end-to-end performance (disk-head time). Optimizing for the right metric in the ML policy improved both introspectability and system performance. We also developed a variant Baleen-TCO, which chooses the optimal flash write rate to optimize our estimate of the total cost of ownership (TCO). This also results in improvements to traditional metrics, reducing IO miss rate by 14% and byte miss rate by 2% (Suppl A.1).

Contributions This paper makes 3 primary contributions: (1) a new cache residency model (*episodes*) that enables a useful comparison point (OPT) and improves ML training effectiveness; (2) ML-guided cache policies that optimize for Disk-head Time and TCO, not hit rate; (3) Baleen, which uses episodes to train coordinated ML admission and prefetching policies, saving 16% in peak load and 17% in (estimated) TCO over our best baseline (Fig 1).

2 Background

2.1 Bulk storage systems in data centers

Tectonic is an example of a bulk storage system, which aggregates persistent storage needs in data centers (e.g., from blobstores and data warehouses). Flash caches are used to reduce the load on the backing HDDs and meet throughput requirements. Other systems have a similar design [16, 35, 42].

Accesses are made to byte ranges within **blocks**. Blocks are mapped to a location on backing HDDs and subdivided into many smaller units called **segments** that can be individually cached. (Tectonic has 8 MB blocks and 128 kB segments.)

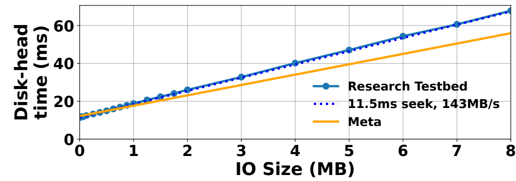


Figure 2: Disk-head Time (DT) for one IO. When a HDD performs an IO, the disk head **seeks** before it **reads** data. For tiny IOs, throughput is limited by *IOPS*; for large IOs, by *bandwidth*. DT encompasses both metrics and generalizes to variable-size IOs.

Upon an access, the cache is checked for all segments needed to cover the request byte range. If any are missing, an IO is made to the backing store to fetch them, at which point they can be admitted into the cache.

Each cluster has 10,000s of storage nodes independently serving requests. Each node has 378 TB in HDDs [35], 400 GB in flash cache, and 10 GB in DRAM cache (37,800:40:1). This paper focuses on the scope of the individual node.

2.2 Bulk storage limited by disk-head time (DT)

At scale, hard disks (HDDs) remain the choice of backing store as they are cheaper by 10X per TB over SSDs [32]. Newer HDDs offer increased storage density, resulting in shrinking throughput (IOPS and bandwidth) per GB as more GBs are served by the same disk head.

Disk-head time (defined in §3.1) on backing HDDs is a premium resource, especially with workloads that are more random than sequential. The mechanical nature of HDDs results in a high, size-independent access time penalty (e.g., 10 ms) for positioning the read/write head before bytes are transferred. With a high read rate (e.g., 5.5 ms/MB), a request could take 10 to 70 ms (Fig 2).

In provisioning bulk storage, peak demand for disk-head time matters most. If the system has insufficient IO capacity, requests queue up, and slowdowns occur. If sustained, clients retry requests and failures occur, affecting user experience. Thus, bulk storage IO requirements are defined by peak load, which in turn affects storage costs.

2.3 Flash caches absorb HDD load but have limited write endurance

Flash caching plays an important role in absorbing backend load, compensating for disk-head time limitations of the underlying HDDs. This setup enables resource-efficient storage for workloads that exceed the throughput requirements of HDDs but which are infeasible to store using flash alone. With the trends towards higher density HDDs and fewer bytes per HDD spindle, flash caches unlock more usable bytes per spindle.

While managing throughput is the primary goal of flash caching, tail latency can improve as a result of reduced backend contention [56]. Flash caches also add flexibility for matching system throughput to ever-growing demand, as it is easier to enlarge flash caches than swap out existing HDDs. When AI training put pressure on storage bandwidth at Meta, the

solution was to add a disaggregated flash caching tier [60].

Flash does not have access setup penalties, but does have wearout that translates into long-term average-write-rate limits. SSD manufacturers rate their drives' endurance in terms of drive-writes per day (DWPD) over their warranty period. Caching is an especially challenging workload for flash, since items will have widely varying lifetimes, resulting in a usage pattern closer to random I/Os than large sequential writes. Items admitted together may not be evicted at the same time, worsening write amplification. Writing every miss into flash would cause it to wear out prematurely. Admitting everything requires up to 492 $\mathrm{MB\,s^{-1}}$ or 43 DWPD for our traces; for an SSD rated at 3 DWPD over 5 years, this means a reduced lifetime of just 4 months (i.e., 14 $\times$ as fast). One solution is SSD capacity overprovisioning, but this can rapidly become a dominant part of the total storage costs [5, 55].

Flash caches leverage **admission policies** (APs) to decide if items should be inserted into the flash cache or discarded, and have simple eviction policies (LRU, FIFO) to minimize write amplification [5]. Like eviction policies, admission policies weigh the benefit of hits from new items against lost hits from evicted items. They must also weigh the write cost of admitting the new item against other past or future items. Policies have an *admission threshold* that can be varied to achieve the target flash write rate. We provide some examples.

- **CoinFlip (baseline)** On a miss, segments for an access are either all admitted, or not at all, with probability p. This simple policy does not need tracking of past items seen.
- **RejectX (baseline)** rejects a segment the first X times it is seen. Past accesses are tracked using probabilistic data structures similar to Bloom filters. We use $X = 1$ and vary the window size of past accesses to achieve the desired write rate. Both Meta [5] and Google [55] used this prior to switching to more complex policies.
- **ML admission policies** use offline features to make decisions in addition to online features such as past access counts. An ML model can be trained offline based on a trace (as we do), or online using reinforcement learning.

2.4 Challenges in flash caching

Challenges in flash admission Flash admission policies are difficult to design for many reasons. DRAM caches do not need admission policies as they can defer decisions to the eviction policy, which has the advantage of knowing the item's usage while in cache. Flash caches incur write costs at insertion time, forcing admission policies to decide a priori to optimize the limited write budget. A longer residency better amortizes this upfront write cost. In contrast, the space-time cost of an item is incurred at a steady rate over time in DRAM caches.

Challenges for ML admission We describe 4 challenges:

Correct optimization metric not obvious The right metric is important not only because optimizing it gives better performance, but because it makes the system more robust. Systems practitioners know the importance of using end-to-end metrics such as IO hit rate, rather than cache hit rate (problem: an IO hit can require multiple cache hits) or ML model accuracy (problem: asymmetrical misprediction cost and class imbalance). Yet even optimizing for IO hit rate is still an (easy) misstep, as a policy that increases the IO hit rate but consumes much more bandwidth may result in overall higher DT, and require more HDDs to serve that load.

Asymmetrical misprediction cost Mispredictions consist of false positives (FPs) and false negatives (FNs). A FP incurs a full write cost (reducing writes left for true positives), and time in cache. FPs have a large performance impact since given the limit on flash writes. With an FN, a hit is lost but the policy may have further chances to admit the item. These lost hits are insignificant for popular items, but have an outsized impact on items with only a few potential hits. There is a long but heavy tail of such items; our traces show many admitted items with 5–8 hits (Fig 20 of Supp A.8). Policies trading off too many FNs for FPs suffer a performance hit [55].

Class imbalance Since most items will not be admitted (94% in our experiments), true negatives (accesses that should not be admitted) far exceeds the number of true positives (accesses that should be admitted). Indeed, we observe that while ML admission policies may achieve a high ML accuracy, this does not always translate into a high cache hit rate. We found typical solutions (oversampling, undersampling, and sample weights) ineffective at countering the extreme imbalance.

APs operate only on misses For an ML policy, it makes sense to train only on accesses in a trace that result in misses, rather than all accesses in the trace. However, this requires an online simulation to determine which accesses are misses, adding additional complexity to training.

Challenges for prefetching policies On a miss, a backend IO must be made to retrieve all missed segments. This IO can be extended and more segments admitted. Done correctly, compulsory misses (when a segment is first observed) are eliminated, reducing disk-head time. However, prefetching mistakes are costly as they consume both writes and extra DT.

Limitations of existing systems Existing works are often:

- **Not modular.** Without a modular design, the system can be oversimplified and miss out on key design considerations [14], or else veer towards too much complexity and be difficult to debug and reason about.
- **Optimizing for intermediate metrics.** Many systems optimize hit rate [8, 13, 14, 22, 37, 43], bandwidth [41, 42] or write rate without considering the larger system the cache is part of. This makes them less performant and robust.
- **Not focused on peak.** Almost all systems report averages, giving less accurate assessments of system performance, as bad performance at peak can be covered up by good (but ultimately unhelpful) off-peak performance. To our knowledge, only one other system evaluates load at peak [42].
- **Not co-designed.** Many systems focus on a single aspect like flash admission [5, 13, 14] or eviction [3, 8, 22, 26, 37,

41–43, 47, 61] without considering the effect of one part on another, in the belief that their benefits will be fully retained when applied with other techniques. To our knowledge, only two other systems evaluate multiple subproblems ([1]: admission and eviction; [56]: admission and prefetching).

3 Exploring potential gains in flash caching

To improve admission, we must first know what "better" looks like. We use *Disk-head Time* as an end-to-end throughput metric to evaluate this. This section describes our decomposition of the flash caching problem, and our attempt at approximating an optimal admission policy (OPT) and a framework (episodes) to evaluate the cost-benefit trade-offs of not just admission policies, but orthogonal improvements such as prefetching.

3.1 Measure Disk-head Time, not hits or bandwidth

We quantify backing store load via *disk-head time* (DT), which is a metric that balances IOPS and bandwidth.

Definition Disk-head Time (DT) is the cost of serving requests to the backend. For a single IO that fetches n bytes, with t_{seek} the time for one disk seek and t_{read} the time to read one additional byte: $DT_i = t_{seek} + n \cdot t_{read}$

Definition Backend load (Utilization) of a time window is the total DT needed to serve misses, normalized by provisioned DT (1 disk-sec per disk per sec): $Util_{DT} = \frac{\sum_i DT_i}{DT_{Provisioned}}$, where

$$\sum_i DT_i = Fetches_{IOs} \cdot t_{seek} + Fetches_{Bytes} \cdot t_{read} \quad (1)$$

DT accurately models throughput constraints of bulk storage systems. DT models both the IOPS and bandwidth limitations of the backing HDDs. (This concept can be extended to other systems with IO setup and transfer costs, such as CDNs.) In our caching setup, we fetch the smallest range covering all cache misses, and normalize DT by HDDs per node to get backend load.

In Fig 3, we validate DT that can be calculated using only two production counters, IO misses and bytes fetched, against system-reported disk utilization on a Meta production cluster in Feb 2023. The peaks line up within 1%, which was surprisingly accurate given the simplicity of this formula (t_{seek} and t_{read} are constants) and the vagaries of production systems (included in the system disk utilization measurements).

DT correctly balances IO misses and byte misses. In practice, $Fetches_{Bytes} \approx Misses_{Bytes}$ (there is a very small difference due to non-consecutive misses). Hence, $\sum DT$ can be interpreted as a weighted sum of IO misses and byte misses, and reducing DT consumed reduces the familiar caching metrics of IO miss rate and byte miss rate.

Conversely, optimizing only the IO miss rate or byte miss rate may result in mistakes made. For example, IO hit rate cannot distinguish these two scenarios though one is better than the other. Consider two blocks, both with 64 accesses. For the first block, each of the 64 segments is requested, one at a time. For the second block, every access requests all 64 segments. While both require the same cache space and save the same IOs, caching the second block saves more DT.

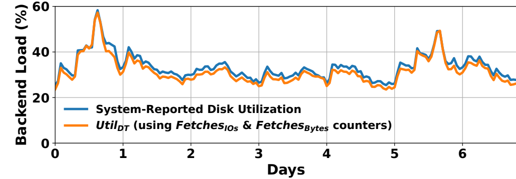


Figure 3: DT validated in production. Our DT formula (plugging counters into Eq 1) matches measured disk utilization (blue) closely. The peak of 58% occurs on Day 0.

Definition Peak DT is the P100 backend utilization ($Util_{DT}$), measured every 10 minutes. The peak refers to the 10-min interval with the highest DT: $PeakDT = Util_{DT}^{P100}$

Peak DT is proportional to the number of backend servers required. System capacity, such as the number of backend servers, is provisioned to handle peak load in systems that need to meet real-time demand. Therefore, to reduce the backend size required, Peak DT should be minimized. This introduces the need for scheduling (i.e., when to spend the flash write rate budget) to prioritize the admission of items that contribute to the Peak DT. As explicitly optimizing admission for the peak introduces significant complexity, we leave that for future work. For this paper, we design our admission and prefetching policies to minimize average DT (and show that they are successful in reducing Peak DT), and optimize for Peak DT in other aspects of the system.

3.2 TCO dominated by backend servers required

In the absence of actual cost numbers, we approximate TCO (total cost of ownership) based on public information. [56] defines TCO as the total cost of HDD reads and written flash bytes, assuming a fixed flash cache size and that other costs (CPU, RAM, power, network) are negligible. We design a similar function, assuming that the cost of HDD reads is proportional to the HDDs required (and Peak DT), and the cost of written flash bytes is proportional to the SSDs purchased in the long run: $TCO \propto Cost_{HDD} \cdot \#HDDs + Cost_{SSD} \cdot \#SSDs$

We calculate relative TCO savings using the Peak DT saved with our baseline AP RejectX ($PeakDT_0$), and relative to the default target flash write rate ($FlashWR_0$).

$$\text{TCO}_1 \propto \frac{\text{PeakDT}_1}{PeakDT_0} \cdot \#HDDs_0 + \frac{Cost_{SSD}}{Cost_{HDD}} \cdot \frac{\text{FlashWR}_1}{FlashWR_0} \cdot \#SSDs_0 \quad (2)$$

This gives us a TCO function based on a policy's Peak DT ($PeakDT_1$) and the flash write rate chosen ($FlashWR_1$). (See App A.4 for a line-by-line derivation.) The skewed ratio of HDD to SSD capacity (945:1 [35]) means that SSD cost is a fraction of TCO (3% on our workloads). Hence, reducing Peak DT (and HDDs needed) is key to reducing TCO.

3.3 Decomposing the caching problem

We define the caching problem as determining which times we should fetch, admit, and evict each segment to minimize

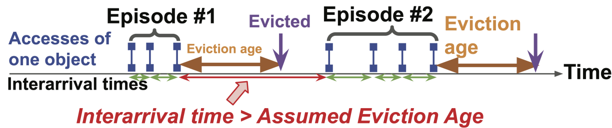


Figure 4: An episode is a group of accesses during a block's residency. Accesses (in blue) are grouped into two episodes as the interarrival time (in red) exceeds the assumed eviction age.

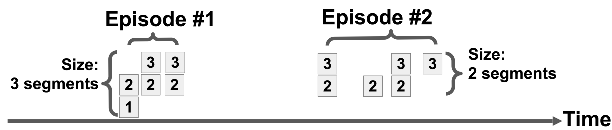


Figure 5: Episodes span space (measured in segments) in addition to time. An episode's size is the smallest number of segments required to be admitted to get all possible hits within an episode. OPT-Range (§ 3.6) is (1,3) and (2,3) respectively.

the backend's DT given a flash write rate limit.

We propose a heuristic decomposition of this problem into three sub-problems: admission, prefetching, and eviction. This makes it easier to reason about the optimal solutions to each sub-problem and the training and behavior of ML solutions for each part. Making ML solutions easier to train, understand, and debug mitigates production engineers' common criticism of their blackbox nature [40].

Admission: Whether to admit something into cache in anticipation of future hits that reduce DT. Here, we trade off the disk-head time saved against the write rate used from caching an item. We model this as a binary classifier, where misses are admitted if the output probability exceeds the policy threshold. We also considered regression models (e.g., predicting no. of expected hits). Such models eliminate the threshold parameter, but we found they perform worse end-to-end, perhaps because their loss functions incentivize performance at all thresholds (write rates) rather than just those at the boundary.

Prefetching: Whether to prefetch extra segments outside the current access range (which was a miss). Here, we trade off DT saved from hits on the first accesses against the additional time spent in cache, and for incorrect prefetches, the DT wasted and the opportunity cost of the wasted flash write rate. We further decompose the prefetching problem into a) deciding what segments to prefetch and b) when to prefetch (whether the expected benefit exceeds the cost, taking into account the possibility of mispredictions).

Eviction: Which segment in the cache to pick for eviction upon an admission. Here, one can employ existing approaches for non-flash caches, including ML-based policies. Here, we employ a simple eviction policy (in our case, LRU) as is used in production systems, leaving ML-based flash-aware eviction policies for future work.

3.4 Episodes: an offline model for flash caching

We devised an offline model for flash caching for efficient evaluation of flash caching improvements, and to facilitate the training of ML-based policies. This model revolves around *episodes*, which are defined as:

Definition An **episode** is a sequence of accesses that would be hits (apart from the first access) if the corresponding item was admitted. It is defined on a block (the rationale being that a cache hit only occurs if all segments are present in cache).

An episode may span multiple segments, and as shown in Fig 5, an episode's size is the number of segments needed to cache it. This leads naturally to a formulation for prefetching. (An important distinction between episodes and block-level LRU analysis is that different episodes for the same block can have different sizes.) An episode's timespan is the length of time between the first access of any segment and the last eviction of a segment.

We generate episodes to aid ML training by exploiting the model of an LRU cache as evicting items at a constant logical time (*eviction age*) after the last access [7, 10, 15, 30]. In an LRU cache, the eviction age is the logical time between an item's last access & eviction. As shown in Fig 4, we group accesses into episodes such that all inter-arrival times within episodes are no larger than the assumed eviction age.

Episodes provide a direct mapping to the costs and benefits associated with an admission, and which corresponds directly to the decisions being made by admission policies. These benefits and costs are associated with an item's entire lifespan in cache, and are not obvious from looking at a stream of individual accesses. Moreover, with flash caching, it is optimal to admit as early as possible in the episode, given that the flash writes required are a fixed cost. By shifting the mental model from interdependent accesses to independent episodes, we can reason about decisions more easily.

Decisions on episodes can be made independently by assuming a constant eviction age. This also allows decisions to be made in parallel. The added pressure on cache space via an admission is accounted for via downward pressure on the eviction age. We determine an appropriate eviction age using simulations that measure the average eviction age. In reality, the eviction age is not constant and varies with cache usage over time. One approach deals with this by calculating policies for a wide range of possible eviction ages [55]. However, we find that in terms of end-to-end performance, Baleen is not sensitive to the assumed eviction age (typically 2+ hours) as long as it is not extremely low (e.g., seconds to minutes).

The episode model also allows for an efficient offline analytical analysis of policies via Little's Law. Given the arrival rate and assumed eviction age, we can estimate the cache size required, and set the eviction age such that the analytical cache size is equal to the cache size constraint. While this is much more efficient than an online simulation and is useful to explore a greater range of parameters than is possible with online simulation, the numbers will differ from simulated ones as the cache size constraint is not enforced all the time, only as a long-term average.

Admission policies can be viewed as partitioning these

episodes into those admitted and discarded. This can be done via scoring episodes and ranking them by score, and we elaborate on this in the next section.

3.5 OPT approximates optimal online AP

Using episodes, we can devise an admission policy (AP) for online simulation that approximates the optimal AP using offline information from the entire trace. First, each block's accesses are grouped into episodes using an assumed eviction age. Second, all episodes are scored and sorted. Last, the maximum no. of episodes are admitted such that the total flash writes required do not exceed the write rate budget. During online simulation, accesses will be admitted if they belong to episodes marked as admitted during the offline process. OPT scores each episode to maximize on the DT saved if admitted and to minimize its size (flash writes required to admit): $Score(Episode) = \frac{DTSaved(Episode)}{Size(Episode)}$

3.6 Prefetching: what and when?

Episodes are also used to design our prefetchers and generate OPT labels for prefetching. By default, on a miss, the smallest IO that covers all missed segments is made, i.e., no prefetching occurs. It is possible to extend this IO and preemptively admit more segments. If done correctly, this reduces the total no of IOs needed and thus reduces DT.

Prefetching the correct segments is important to achieve a reduction in DT given a write bound. With imperfect admission policies, predicting a confidence value is necessary to balance the risk of real prefetching costs against possible benefits. Otherwise, prefetched segments compete with segments admitted from misses and drive up write rate while not reducing DT, meaning an overall reduction in DT for the same write bound. Note that the costs and benefits of prefetching must be evaluated against the opportunity cost of using writes for admission of missed blocks instead.

Deciding *when* to prefetch Fetching insufficient segments results in minimal or no DT reduction. On the other hand, fetching excess segments results in a high write rate. To balance these trade-offs, we need to know our confidence in our range prediction.

For instance, prefetching the entire block *on every miss* will result in an overall IOPS reduction given write rate constraints. A blunt method to increase precision is to prefetch *on every 2nd miss* or *on every partial IOPS hit* (when some but not all segments in an access return a hit). This indicates that part of the block was admitted to cache. For OPT prefetching, we prefetch on *OPT-Ep-Start*, the start of the episode as determined by the episode model.

Deciding *what* to prefetch: Whole-Block, OPT-Range The straightforward choice is to prefetch the entire 8 MB block (*Whole-Block*). However, the resultant write rate is too high, making it infeasible unless combined with prefetching on every partial IOPS hit. To evaluate how well we could perform given offline information from the whole trace, we introduce

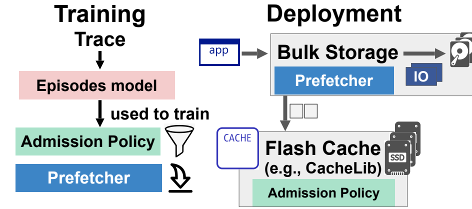


Figure 6: Architecture. An admission policy in CacheLib decides whether to admit items into flash. Prefetching (preloading of data beyond current request) takes place in Tectonic.

OPT-Range, which uses the generated episodes to determine an optimal range of segments to prefetch. OPT-Range is the minimal range of segments that covers all accesses in an episode. For the episodes in Fig 5, OPT-Range is (1,3) for Ep 1 and (2,3) for Ep 2. Whole-Block always fetches (1,64).

4 Baleen Implementation

We describe how Baleen provides episode-based solutions to two problems: how to train an **ML-based admission policy**, and using **prefetching** to improve beyond admission policies.

4.1 Training Baleen's ML admission policy

Episodes generated from the trace are used to train an admission policy, as shown in Fig 6. The policy is a binary classification model. We describe: 1) how we generate training data and labels from episodes, 2) what features and architecture we use for the ML admission model, 3) how we determine appropriate values for training parameters (assumed eviction age, admission policy threshold) through an iterative loop, and 4) how we implement ML admission in CacheLib.

Features Baleen's admission policy utilizes a total of 9 features, grouped into offline metadata and online usage counts.

Metadata features are provided by the bulk storage system and supplied in the trace. These metadata features identify the provenance of the request (namespace, user) and indicate whether the block is tagged as temporary (e.g., as a result of a JOIN) or permanent. Feature cardinality is less than 100 for namespaces and less than 200 for users. Both features are associated with the system user (internal service) executing the request rather than an end user. These features are often the same for accesses to the same object and almost always the same for accesses belonging to the same episode. These features are provided per IO and thus the same for all segments.

Online dynamic features (times the item is accessed in the last 1,2, . . . 6 hours) change with every access. This can be measured at the block or segment level. For Baleen, we record both the number of IOs for each block and the cumulative segment accesses for each block to use as features. For each workload, a simple simulation is done on the training set (the first day) to collect these dynamic features. We do not use individual segment counts as features, as this would add 64 features without an appreciable increase in performance.

Modeling admission as binary classification We admit misses if the classifier's output probability exceeds the policy

threshold. We also considered regression models (e.g., predicting no. of expected hits). Such models eliminate the threshold parameter, but we found they perform worse end-to-end [14], perhaps because their loss functions incentivize performance at all thresholds and not just those at the boundary.

Training data and label generation The goal is to differentiate episodes at the decision boundary, which tend to have few accesses. Learning to identify these episodes is hard but important as they are significant in aggregate. To avoid a training bias towards popular but easy-to-differentiate episodes, only the first 6 accesses from each episode are incorporated into training data. Baleen learns to imitate OPT, and the binary labels are determined by whether that episode, based on its score, would have been admitted under OPT.

Converging on Eviction Age, Policy Threshold Parameters We repeatedly run the offline episode model and online simulation in a loop to converge on values for assumed eviction age (EA) and admission policy threshold. Recall that episodes are generated with an assumed EA. These episodes are used to train models, which are used in an online simulation where the average EA can be measured. We initialize assumed EA to an arbitrary value of 2 hours and repeat episode generation, model training, and online simulation until the assumed EA converges on the average EA from an online simulation. Within each loop iteration, there is another nested loop to find the correct admission policy threshold that results in the simulation achieving the target flash write rate. This inner loop aims to offset the small differences between offline analysis and a higher-fidelity online simulation.

Online flash caching simulator The training loop mentioned in the prior paragraph requires an online simulator to be run multiple times. We developed a Python simulator to accurately estimate CacheLib performance without doing the actual heavy lifting. This is an approach taken by other ML for Systems projects [44]. This lightweight simulator is easier to include in a ML training pipeline, and takes as input a Tectonic trace and measures many end-to-end metrics (e.g., average eviction age, Peak DT) that cannot be obtained from offline episode analysis. Having the training setup be Python-centric aids in faster prototyping, ease of use by data scientists, and ease of integration with existing ML training pipelines.

Gradient boosting machines (GBM) We chose to use GBMs as they are fast and have some inherent tolerance to overfitting and imbalanced classes. Compared to deep neural networks, they are far more efficient and are well-proven to run within the latency requirements of a production caching system [5]. Practitioners also find them easier to understand, given that they are based on widely-understood decision trees.

Adding a ML admission policy to CacheLib The open-sourced version of CacheLib supports flash admission policies, but does not include a mechanism for storing and supplying features to ML admission policies. We describe how this may be done. For the static metadata features, they can be embedded as part of the item payload. Since payloads are a few MB on average, storing the features (less than 1 kB) in this way does not impose any significant overhead. To provide the dynamic features, counts of accesses are tracked in CacheLib using a count-min-sketch data structure (similar to bloom filters, but with counts). Each data-structure holds the count for approximately one hour, with a queue of 6, such that we have counts at hour-level granularity for the last 6 hours.

4.2 Training Baleen's Prefetcher using episodes

Models are trained to solve two subproblems: what to prefetch, and when to prefetch.

Learning what to prefetch: ML-Range We need a ML model that predicts a range of segments for prefetching. We do this by training the model to imitate *OPT-Range*, the smallest range of segments needed for all accesses in an episode to be hits (defined in §3.6). We use the same features as the ML admission model, but add size-related features (access start index, access end index, access size). We train two regression models to predict the episode range start and end. Each episode is represented once in the training data, with only episodes that meet the score cutoff for the target write rate included.

Learning when to prefetch: ML-When Mispredictions by the ML admission policy and in ML-Range can easily cause prefetching to hurt instead of help. In reality, the expected benefit will be lower than OPT prefetching and the cost can only be higher. DT saved from prefetching ML-Range may not be realized (which we call underfetch, see Eq 3a). Further, prefetching mispredictions are costly in terms of DT consumed to fetch unused segments (which we call overfetch, see Eq 3b) and the opportunity cost of flash writes used to store them.

ML-When aims to address this by **excluding episodes that do not have a high probability of benefiting from prefetching**. In particular, it hedges against the broader effect of prefetching on eviction age by requiring that the marginal DT gained from ML prefetching ($PFBenefit^{ML}_{eps}$, Eq 3c) be larger than ϵ (ML-When label, Eq 3e). ϵ is a proxy for the unknown broader opportunity costs of flash writes and cache space, and set to 5 ms (for comparison, an IO seek is 12 ms).

$$UF : underfetch = true \text{ if ML-Range} \subset \text{OPT-Range} \tag{3a}$$

$$OF : overfetch = DTUsed(\text{extra segments}) \tag{3b}$$

$$PFBenefit^{OPT}_{eps} = DT^{NoPrefetch}_{eps} - DT^{OPT-Range}_{eps} \tag{3c}$$

$$PFBenefit^{ML}_{eps} = \begin{cases} 0 & \text{if underfetch} \\ PFBenefit^{OPT}_{eps} - OF & \text{otherwise} \end{cases} \tag{3d}$$

$$\text{ML-When}(eps) = PFBenefit^{ML-Range}_{eps} > \epsilon \tag{3e}$$

Prefetching is implemented in CacheLib applications Every request to the bulk storage system references a block in the backing store and a byte range within that 8 MB block. Each request is translated by the application into (potentially multiple) CacheLib segment-level requests. CacheLib is not aware that segments may belong to the same block.

Thus, prefetching must be implemented by the application issuing requests against CacheLib, which is the bulk storage

system in our case (Fig 6). On each client request, Baleen's prefetcher will be triggered after the application has queried CacheLib and found out whether segments are hits or misses. Thus, the prefetcher has access to the client request metadata and knows how many requested segments were present in cache. On a miss, the application makes a request to the backing store, giving the prefetcher a chance to fetch extra segments and insert those into cache.

4.3 Optimizing for Peak DT and TCO

Baleen-TCO We designed a variant of Baleen, Baleen-TCO, that optimizes our TCO function (Eq 4b) by simulating Baleen over a range of flash write rates to get the respective Peak DT. Baleen-TCO then chooses the optimal flash write rate to minimize the TCO function.

Optimizing for Peak DT. Prefetching is key to Baleen's performance on most workloads, but on some workloads, ML-When is not aggressive enough as it optimizes for the mean, not Peak DT. To correct for this, we allow Baleen to choose another prefetching option per workload (e..g, *ML-Range on Partial-Hit*) if it is better at reducing Peak DT in training.

5 Evaluation

This section evaluates and explains Baleen's effectiveness in reducing backend peak load & TCO for 7 real workload traces.

5.1 Experimental setup

We evaluate Baleen using a testbed and a simulator. We validate both with counters from production deployments. Our key results use simulation runs, but we validate individual points (e.g., Fig 11). We explain our setup, workloads, metrics, and the flash write rate and cache size constraints used.

ML training setup We wrote a Python module that generates episodes and trains the ML models. This plugs into a Python simulator for CacheLib we developed for training and prototyping (§4). We validate this Python simulator against testbed and production (§5.2). The episode module takes in a trace and returns the ML models. We then run simulation loops to converge on an assumed eviction age and admission policy threshold. LightGBM [19] was used for training and inference, with 500 rounds of boosting and 63 leaves.

Implementation in CacheLib We implemented support for ML admission and prefetching policies.We emulate calls to Tectonic so that every miss issues a real IO of the right size against HDDs, and measure the wall-clock time as DT consumed. Static features are stored in the CacheLib payload, while history counts are tracked by CacheLib. We use CacheLib's region-based LRU with a region size of 142 kB.

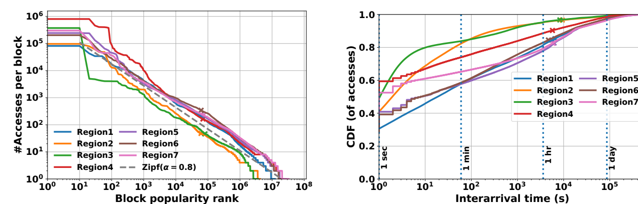
Overhead Baleen's overheads are low in the context of caching for bulk storage systems. *CPU overhead:* Baleen adds 4 inferences per IO miss (admission, start & end of ML-Range, ML-When). The system is limited by the latency of disk IOs upon misses (10–70ms per IO) rather than ML inferences (30 microseconds per inference). Even when replaying a trace at full speed, CacheLib only contributes a small fraction of overall system CPU utilization (5% of the 16-core CPUs in our testbed) because it is waiting for disk IO, and thus using ML policies only translate to an additional 1% increase in overall CPU usage. *Metadata overhead:* Baleen also stores static metadata features in the payload (<1kB), but as payloads are at least 128KB, this overhead is not significant (<1%).

Hardware The Tectonic production setup used to record traces and counter values has a 400 GB flash cache, 10 GB DRAM cache and 36 HDDs. Our academic testbed uses enterprise-grade hardware, but with less HDDs per node and thus a proportionally smaller cache size (see Suppl A.9).

Table 1: Key statistics of traces.

Dataset	Req Rate (s^{-1})	Access size (MB)	CMR[1]	OHW[2]	Admit-All Writes (MB/s)
Region1	244	3.41	18%	54%	316
Region2	106	2.85	39%	83%	121
Region3	139	2.42	19%	48%	45
Region4	406	2.87	14%	53%	280
Region5	364	2.62	18%	59%	480
Region6	404	2.74	14%	55%	478
Region7	426	2.23	17%	62%	492

[1] CMR (Compulsory miss rate): ratio of blocks to accesses;
[2] OHW (One-hit-wonder): % of blocks with no reuse.



(a) Block popularity (log-log). **(b)** Interarrivals to same block.

Figure 7: Block popularity and access interarrival. In a, lower values of α indicate it is harder to cache, and × denotes 400 GB. In b, × denotes eviction ages for Baleen at 400 GB & 3 DWPD.

Workloads We report results for 7 production traces from Meta. Each workload sampled production traffic from a Tectonic [35] cluster, which can span an entire datacenter, including traffic for hundreds of applications. Traces were recorded in 3 different years. The popularity distribution of blocks (Fig 7a) fit a Zipf($\alpha = 0.8$) distribution, where the i-th most popular block has a relative frequency of $1/i^{\alpha}$. Fig 7b shows the interarrival time distribution, with the converged eviction age for Baleen marked with crosses. For all traces, less than 20% of interarrival times exceed the converged eviction age. The majority of blocks are the maximum size (8 MB) with averages of 5.1-6.8 MB across traces, but most accesses are only a fraction of the block with the median access less than 2 MB. Full details are available in Supp A.8.

Each trace is sampled from an entire cluster (each numbering

thousands of nodes). A fraction of traffic is sampled at every node and the resulting samples aggregated to get a trace. The sampling rate and number of nodes are recorded, and used in further downsampling of the trace. For quicker simulation runs, the trace is sampled on the block key space, with each block weighted by number of accesses, with the cache size scaled down proportionally. A train-test split is performed on the time dimension, i.e., the first day of each workload is used as training data, with the remaining days used for testing.

Metrics and Assumptions The savings from using Baleen are dominated by the degree by which it reduces the no. of HDDs required to handle peak load. Therefore, our evaluation focuses on Peak DT (see § 3.1). To aid comparison across traces, we normalize each policy's Peak DT by the Peak DT required with no cache. We also show estimated TCO savings over RejectX (using Eq 4b).

Testbed results (used to validate our simulator at a fixed flash write rate) used 1-5% samples (maximum sample rate is 5%, limited by the ratio of HDDs (2:36)). Simulator results used 0.1-5%-traces. The sample percentage is higher for smaller workloads. We scale to a 400 GB-equivalent flash cache and our target flash write rate.

Baleen accounts for differences in hardware (HDDs, SSDs) via the target flash write rate and constants in the TCO & disk-head time formulas. For flash, the pertinent characteristics are those affecting endurance (and thus write rate). Fig 10 show Baleen performing at different flash write rates and cache sizes. Baleen-TCO also allows for a different flash-to-HDD cost ratio to be substituted in. For HDDs, our simulations assume a constant average seek time and bandwidth in the DT formula (Eq 1). These parameters vary minimally across disks, as illustrated in Fig 3 (simple formula closely matches actual disk utilization in production). Baleen includes a small benchmark to measure these constants for a given disk.

Baselines We compared Baleen to 4 baselines: RejectX, CoinFlip, and two state-of-the-art ML baselines, Flashield [14] and CacheLib [5]). We focus on RejectX as it is publicly available and has been chosen over state-of-the-art ML models in industry. The CacheLib ML policy addresses Flashield's limitations (see §5.2) and uses non-episode-related features.

5.2 Baleen reduces Peak DT over baselines

Fig 1b shows Baleen reduces Peak DT over RejectX by an average of 12% across all traces for a fixed target flash write rate. Fig 9 shows this ranges from 5% to 29% across the traces. Region1, Region3 and Region4 derive most of their gains from prefetching.

Flashield is not shown in the graphs as it failed on half the trace samples due to insufficient training data (more details in Suppl A.5). If we consider only workloads Flashield could train a model on, Baleen outperformed Flashield by 18%.

Validation of simulator and testbed Fig 11a shows us validating Baleen on our simulator against Baleen on our testbed. Further, we took the additional step of showing that our testbed is consistent with production counters, and show it matches closely (Fig 11b).

Training on episodes (instead of accesses) is essential to ML prefetching Episodes make it easier to reason about flash caching and was key to designing both OPT and ML prefetching. We also found that in the absence of episodes, others in the literature devised ad-hoc sampling heuristics that would achieve the same goal of avoiding ML training bias towards popular objects [41]. In addition, we quantify the benefit of episodes by comparing Baleen to an earlier ML admission policy that did not use episodes. Adding prefetching to the non-episode-based ML admission would cause it to perform worse than without prefetching.

Benefits consistent at higher write rates and larger cache sizes Fig 10 shows that Baleen allows for a reduction in cache size by 55% while keeping the same Peak DT as RejectX, or alternatively a reduction in Peak DT equivalent to a 4X increase in cache size. As expected, increasing write rate or cache size has diminishing returns in reducing Peak DT. Also, the different admission policies (without prefetching) start to converge, indicating that admission by itself is insufficient to drive further reductions in Peak DT. Graphs for all 7 traces are available in Supp A.11.

5.3 Baleen-TCO chooses optimal flash write rate

Fig 8 shows Baleen-TCO reducing TCO by 17% over CacheLib-ML and 18% over RejectX. Workloads have different optimal flash write rates; Baleen-TCO picks the best flash write rate for each, as illustrated in Fig 12. If a constant flash write rate target is used, Baleen is able to reduce TCO by 14% over RejectX. (Thus, Baleen-TCO saves an additional 4% over Baleen with a fixed write rate). Flash writes account for 2% to 5% of TCO (3% on average).

5.4 Prefetch selectively, in tandem with admission

We show both ML-Range and ML-When are effective in reducing Peak DT over static baselines, and contribute to Baleen's robustness across the multiple traces. We also show that prefetching must be paired with a good admission policy; if not, the same prefetching policy can hurt rather than help.

ML-Range outperforms no prefetching and fixed range prefetching. Using ML to decide what to prefetch (ML-Range) saves 16% of Peak DT over no prefetching, and 4% over a simple baseline (All on Partial Hit) (Fig 13). Baleen admission is used in all cases, with only the prefetching policy varied. We note this comes with a small increase in Median DT.

ML-When helps Baleen discriminate between beneficial and bad prefetching. ML-When expresses Baleen's confidence in the quality of its ML-Range prediction. A general challenge with prefetching is that one is predicting without a direct signal (such as a miss in the case of admission). If used indiscriminately, prefetching can hurt rather than help. This is best illustrated by how prefetching ML-Range on Every Miss is worse than no prefetching in Fig 14. Prefetching only on ML-When or on Partial-Hit consistently does better than

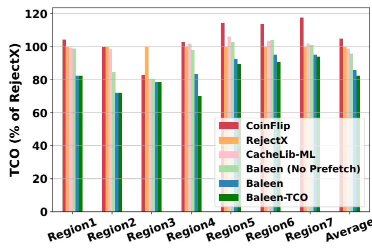


Figure 8: Baleen-TCO reduces TCO.

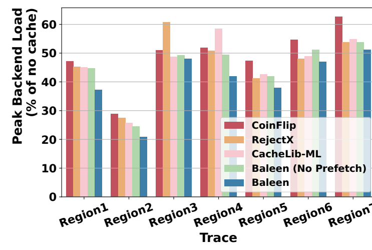


Figure 9: Baleen reduces Peak DT.

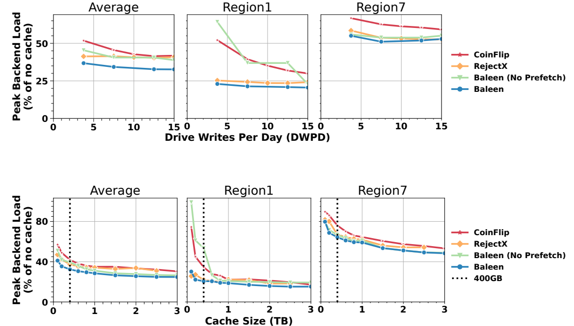
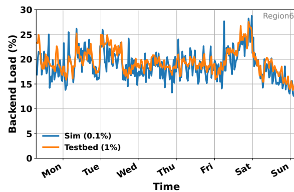
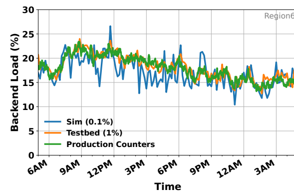


Figure 10: Benefits at higher write rates & cache sizes.



(**a**) Sim vs Testbed, Baleen



(**b**) Testbed vs Prod, RejectX

Figure 11: Validation of simulator and testbed.

both no prefetching and prefetching on every miss across all traces.ML-When performs better on 2 traces (Region2, Region7) and Partial Hit on the remaining 5.

Poor admission decisions lead to poor prefetching ML prefetching reduces Peak DT most when paired with a good admission policy like Baleen. With RejectX, prefetching is less helpful or even hurts (in Region7). Thus, the Baleen admission policy is important to the performance of prefetching despite not always reducing Peak DT by itself. Adding prefetching to CoinFlip yielded results similar to RejectX.

5.5 Optimizing the right metric: Peak DT

Optimizing for IO hit ratio can be misleading as doing so is optimal for reducing seeks, not total disk-head time. Policies that do so may reduce IOs at the expense of increased bandwidth, which can be a net loss in bandwidth constrained systems. For example, for the prefetching option "ML-Range on Every Miss" from Fig 14. relative to no prefetching, the mean DT used ratio increases from 67% to 73% despite the IO hit ratio increasing from 46% to 47%.

DT during peak periods Most of the reduction in Peak DT comes from eliminating seeks rather than read time, often through prefetching. Certain traffic patterns affect some policies more, which is why the DT peaks for different policies can differ. In particular, Baleen's peaks occur when prefetching is not beneficial. We show further analysis in Supp A.3.

5.6 Other ML-guided cache results/experiences

Baleen is the end result of substantial exploration and experimentation with ML for caching, including negative outcomes from which we drew lessons and see unrealized potential. This section shares and quantifies these lessons.

GBM better than deep models (Transformer & MLP) We compared GBM to more complex ML architectures (a Transformer-based architecture we designed and MLP). We found that GBM performs best (0.2% better than Cache Transformer), despite only having features for the current access. A challenge we faced when training these deep models were the highly imbalanced classes. Details are in Supp C.

Explicitly optimizing Peak DT Fig 15 shows DT varying over time, with a peak-to-mean ratio of 2. A policy wanting to optimize Peak DT should be aware of the current load level and able to adapt to it. We performed a simple extension where we only admitted to the cache during periods of high load. We found that while this saved flash writes, it did not reduce Peak DT. This suggests that more fundamental changes (e.g., scoring episodes by their usefulness in reducing Peak DT) will be required to optimize explicitly for peak load.

Baleen benefits from size-awareness. An earlier ML model required explicit size-awareness for a 5%-savings in mean DT. Baleen learns it implicitly if size-related features are supplied.

Gap between Baleen and OPT Fig 13 shows a remaining gap of 16%, indicating significant room for improvement. Episode-based analysis shows 9% of DT is lost to late admissions (i.e., where episodes are admitted after the first access). We observed Baleen learning to reject almost all items on the first access (a behavior similar to RejectX). Many training examples shared identical features (on the first miss) but had different labels. Baleen thus predicted the most probable label for each feature set (i.e., Bayes Optimal classifier behavior). Since dynamic, history-based features cannot differentiate unseen items, we hypothesize that better metadata features are required to distinguish the few true positives.

Segment-aware admission & prefetching Baleen operates at the block level and can only choose to admit or reject the entire access range, rather than individual segments (unlike RejectX). Episode-based analysis showed a potential reduction of DT by 11%. However, we were unable to realize this.

Prefetch on PUT This would yield an additional hit on the first-ever access to the item. However, this is difficult as many written blocks are not touched again for the duration of our

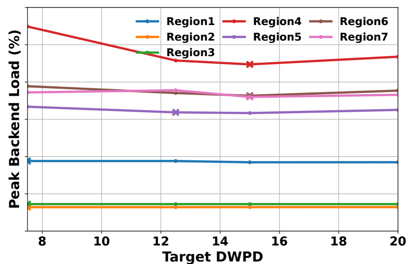


Figure 12: **Baleen-TCO chooses higher flash write rates when needed to lower peak backend load (and TCO).** × denotes the optimal flash write rate for that workload.

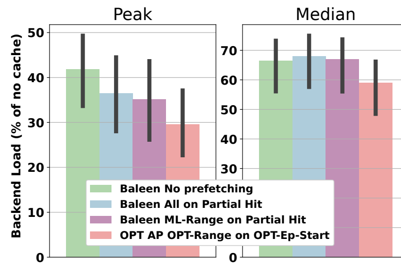


Figure 13: ML-Range saves Peak DT. ML-Range outperforms the baseline (whole block) and No Prefetching at the expense of Median DT.

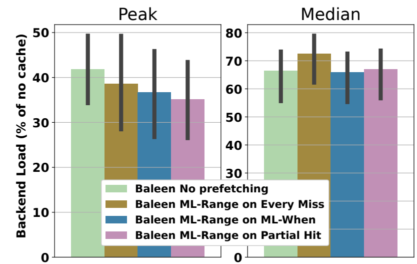


Figure 14: Choose *when* to prefetch. Indiscriminate prefetching (on Every Miss) can hurt. Using ML-When or Partial Hit reduces Peak DT without compromising Median DT.

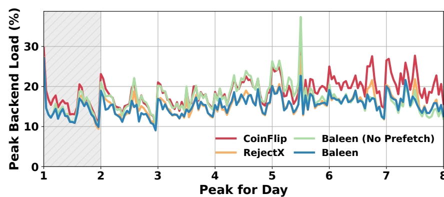


Figure 15: Testbed backend load on Region1. Day 1 (shaded) is used as training data. Peak is on Day 5 and is lowest for Baleen.

traces, meaning that a classifier must be extremely accurate or else incur costly false positives. This could work for other workloads with higher incidences of read-after-writes.

Early eviction If items could be evicted immediately after their last access in an episode, instead of waiting to leave the cache, this would eliminate dead time and result in a greater effective cache size. Episode-based analysis showed mean DT could potentially be reduced by 11%. However, our current ML models are not accurate enough to realize this.

6 Lessons from deploying ML in production

We summarize a few lessons gleaned from 3 years of deploying ML in production caches at Meta.

Optimizing the wrong metric is an easy misstep. Our initial prototypes for prefetching and admission increased IO hit rate, but was actually worse for DT. To overcome this, we redesigned our ML admission policy and introduced a prefetching confidence prediction (ML-When). Picking the right end-to-end metric is important.

ML model performance does not always translate to production system performance. The same algorithm performs differently when moved from offline to online settings, and again when moved from development to production environments. Evaluation in production is slow (many days needed to collect data in real time) and laborious (restarts, aborts, debugging). This makes it challenging to tune thresholds and evaluate improvements to ML policies. The plethora of directions makes it hard to decide on the best path forward without extensive exploratory research. This motivated our episodes model that allows for the principled design of ML policies that can directly optimize systems metrics like DT under write rate constraints, and quickly evaluate the end-to-end impact of hypothetical improvements without the effort to implement them in production or debugging unrelated production noise.

Rethink use of DRAM in flash caching. The typical use of DRAM is as a small cache before [5, 14] (or after [55]) flash, with admission decisions made on DRAM evictions. We moved the admission policy from post-DRAM to pre-DRAM, with minimal impact on end-to-end metrics. The initial motivation was saving DRAM bandwidth, as this became a bottleneck with Admit-All rates near 500 MB/s (Table 1). The impact was small – while a DRAM cache may appear to absorb hits, it is simply stealing them from the flash cache. Since DRAM eviction ages (a few seconds) are so much shorter than flash (2+ hrs), almost every item worth caching needs to be in flash. Further, the write costs of an item are proportional to its size, and any potential avoidance of flash writes is limited by how small the DRAM cache is (2.5% of flash cache). Flexible placement of the admission policy enables optimizations such as prefetching, which must be done prior to inserting into the topmost cache. In summary, we need to find better uses for DRAM than simply adding it before a flash cache.

ML-based caching should aim for encapsulation of ML, caching, and storage. Designing bespoke ML for caching solutions requires coordination between ML experts (for model training), caching experts (for integration), and the storage backend owner (for deployment and monitoring). This involves one more area of expertise than most other ML for systems problems. There is no clear path to single ownership of the problem, making it difficult to sustain over time. It is hard for a service owner to prioritize spending engineering resources to aid the design phase of unproven ML solutions. Baleen provides an analytic framework that ML experts could optimize DT on without requiring caching expertise. Designing ML models around episodes makes it easier for caching experts to reason about. Having the DT formula correspond closely to measured DT (Fig 3) in production assures caching and storage experts that a reduction in calculated DT will translate to a drop in disk utilization. Further, with setups that are

tightly-coupled by hand and not automatically, performance regressions may occur as systems and workloads change. Models often performed the best when they were first deployed and slowly regressed over time even with retraining using the same set of features. In contrast, Baleen was designed primarily using traces from 2019 but also demonstrates improvements on traces from 2021 and 2023.

7 Additional related work

Production flash caching systems CacheSack [55, 56] optimizes admission policies for the flash cache in front of Google's bulk storage system, Colossus. This design shares Baleen's objectives of co-optimizing backend disk reads and flash write endurance. CacheSack partitions traffic into categories using metadata and user annotations, assigning probabilities to each of 4 simple admission policies for each category by solving a fractional knapsack problem. This offline approach has slower reaction times than Baleen, and only solves the admission problem. Meta's Tectonic bulk storage system uses a CacheLib-managed flash cache, with an ML admission policy that does not use episodes and does not perform prefetching. Section 5.6 shows that this approach is significantly less effective than Baleen. Kangaroo [31] improves CacheLib's small object cache, and is orthogonal to Baleen, which improves performance for large objects. Amazon's AQUA [2] also fills a similar role for Redshift (data warehouse), acting as an off-cluster flash caching layer with S3 as the backing store. Bulk storage systems backed by HDDs and fronted by cache servers can also be found at Alibaba Cloud [23] and Tencent [58].

Non-ML flash admission policies CacheLib [5] is Meta's general-purpose caching library and includes random and RejectX admission policies for flash caches. Section 2 discusses RejectX. Section 5 extensively compares Baleen to random (CoinFlip) and RejectX. LARC [18] is equivalent to RejectX and was the default admission policy used at Google prior to CacheSack. TinyLFU [13] proposed a frequency-based admission policy that leverages probabilistic data structures for compact history representation. Baleen adds ML, size-awareness, disk performance goals, and prefetching over TinyLFU.

ML-based flash caching policies Flashield [14] addresses the lack of information on flash admission candidates by putting them in a DRAM buffer first. The item's usage history is used to generate features for a support vector machine classifier. However, we found this approach infeasible as DRAM lifetimes are too short in practice (see Supp A.5). More targeted applications of ML aim to exclude one-hit-wonders [48] or items that have no reads [59]. Reinforcement learning has also been used to train a feedforward neural network for admission policies on CDNs, given a broad set of features [20]. Baleen adds more flexible admission policies, size-awareness, disk performance goals, and prefetching over these works. Early work on flash caching focused on flash-friendly eviction policies [36]. Recent work instead uses simpler eviction policies such as CLOCK or FIFO, and leaves the heavy lifting to the admission policy [55]. Smart policies for data placement seek to reduce write amplification [9], and can be used in tandem with Baleen.

Prefetching policies CacheSack [56] incorporated static prefetching policies as choices for their optimization function. [62] implemented heuristic-based prefetching for photo stores, but found significant room for improvement relative to their offline optimal. Others have posed caching as a scheduling problem in the context of streaming video and incorporated aspects of prefetching [27, 38, 46]. In databases, Leaper trains a ML prefetcher to exploit reuse at the key range level [54].

Models for caching and offline optimal Bélády's MIN algorithm is the optimal eviction policy [4]. [41] introduces Relaxed Bélády for eviction which prunes the decision space like OPT does; however OPT makes stronger assumptions valid for flash admission and decides at a higher granularity (see Suppl A.7). Raven [17] is a probabilistic approximation of MIN. [11] sought to extend Bélády to admission with a container-optimized MIN that optimizes hit rate while minimizing flash erasures, but did not provide an online algorithm. Our proposed OPT policy is the only online policy that approximates the optimal flash admission policy, and which can easily optimize an arbitary metric like DT, not just hit rate.

ML for eviction Some policies seek to learn from Bélády, such as LRB which learns a relaxed Bélády [41], and RL-Bélády [51]. A key challenge to using RL is the long delays for rewards. [6] Others seek to go beyond Bélády, such as LRU-BaSE [49]. MAT [52] reduces ML inference overhead by using a heuristic to filter out likely candidates. HALP [42] augments a heuristic with ML for the YouTube CDN. Deep learning has also been applied to learn forward reuse distance with LSTMs [24] and reinforcement learning [50]. [39] uses a support vector machine with features they derived from training an LSTM. [12] proposes that a classical caching policy be run in parallel with ML policies, allowing the implementation to switch to the better-performing policy dynamically. ML-based eviction is orthogonal to Baleen's contribution and cannot control flash write rates.

8 Conclusion

Baleen uses ML to guide both prefetching and cache admission, reducing peak disk time by 16% and TCO by 17% on real workload traces, compared to state-of-the-art non-ML policies. Although applying ML to caching policies is an expected advancement, Baleen's design arose from false-step lessons and a cache residency (episodes) formulation that improves training effectiveness, provides a target (OPT), and exposes the value of ML-guided prefetching. As such, Baleen is an important step forward in flash caching for disk storage.

Acknowledgments

We thank Meta, Google, and the members and companies of the PDL Consortium (Amazon, Google, Hitachi, Honda, IBM Research, Intel Corporation, Meta, Microsoft Research, Ora-

cle, Pure Storage, Salesforce, Samsung, Two Sigma, Western Digital) and VMware for their interest, insights, feedback, and support. We thank the anonymous reviewers and our shepherd, Ali Anwar, for their helpful comments and suggestions. This research is supported in part by NSF grant CNS1956271.

References

[1] Zahaib Akhtar, Yaguang Li, Ramesh Govindan, Emir Halepovic, Shuai Hao, Yan Liu, and Subhabrata Sen. Avic: a cache for adaptive bitrate video. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 305–317, 2019.

[2] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2205–2217, 2022.

[3] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *NSDI*, pages 389–403, 2018.

[4] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[5] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768, 2020.

[6] Daniel S Berger. Towards lightweight and robust machine learning for cdn caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 134–140, 2018.

[7] Daniel S Berger, Philipp Gland, Sahil Singla, and Florin Ciucu. Exact analysis of TTL cache networks. *Performance Evaluation*, 79:2–23, 2014.

[8] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. AdaptSize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 483–498, 2017.

[9] Chandranil Chakraborttii and Heiner Litz. Reducing write amplification in flash by death-time prediction of logical block addresses. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, pages 1–12, 2021.

[10] Hao Che, Ye Tung, and Zhijun Wang. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE Journal on Selected Areas in Communications*, 20(7):1305–1314, 2002.

[11] Yue Cheng, Fred Douglis, Philip Shilane, Grant Wallace, Peter Desnoyers, and Kai Li. Erasing Belady's limitations: In search of flash cache offline optimality. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 379–392, 2016.

[12] Jakub Chłędowski, Adam Polak, Bartosz Szabucki, and Konrad Tomasz Żołna. Robust learning-augmented caching: An experimental study. In *International Conference on Machine Learning*, pages 1920–1930. PMLR, 2021.

[13] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017.

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

[15] Christine Fricker, Philippe Robert, and James Roberts. A versatile and accurate approximation for LRU cache performance. In *2012 24th international teletraffic congress (ITC 24)*, pages 1–8. IEEE, 2012.

[16] Dean Hildebrand and Denis Serenyi. Colossus under the hood: a peek into Google's scalable storage system, 2021.

[17] Xinyue Hu, Eman Ramadan, Wei Ye, Feng Tian, and Zhi-Li Zhang. Raven: belady-guided, predictive (deep) learning for in-memory and content caching. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*, pages 72–90, 2022.

[18] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. Improving flash-based disk cache with lazy adaptive replacement. *ACM Transactions on Storage (TOS)*, 12(2):1–24, 2016.

[19] 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. *Advances in neural information processing systems*, 30, 2017.

[20] Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K Sitaraman. RL-Cache: Learning-based cache admission for content delivery. *IEEE Journal on*

Selected Areas in Communications, 38(10):2372–2385, 2020.

[21] Mathias Lecuyer, Joshua Lockerman, Lamont Nelson, Siddhartha Sen, Amit Sharma, and Aleksandrs Slivkins. Harvesting randomness to optimize distributed systems. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets)*, pages 178–184, 2017.

[22] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. Pannier: Design and analysis of a container-based flash cache for compound objects. *ACM Transactions on Storage (TOS)*, 13(3):1–34, 2017.

[23] Jinhong Li, Qiuping Wang, Patrick PC Lee, and Chao Shi. An in-depth analysis of cloud block storage workloads in large-scale production. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 37–47. IEEE, 2020.

[24] Pengcheng Li and Yongbin Gu. Learning forward reuse distance. *arXiv preprint arXiv:2007.15859*, 2020.

[25] Chieh-Jan Mike Liang, Hui Xue, Mao Yang, Lidong Zhou, Lifei Zhu, Zhao Lucis Li, Zibo Wang, Qi Chen, Quanlu Zhang, Chuanjie Liu, et al. AutoSys: The design and operation of Learning-Augmented systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 323–336, 2020.

[26] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An imitation learning approach for cache replacement. In *International Conference on Machine Learning*, pages 6237–6247. PMLR, 2020.

[27] Jiangchuan Liu and Bo Li. A qos-based joint scheduling and caching algorithm for multimedia objects. *World Wide Web*, 7:281–296, 2004.

[28] Martin Maas. A taxonomy of ML for systems problems. *IEEE Micro*, 40(05):8–16, 2020.

[29] Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. Variance reduction for reinforcement learning in input-driven environments. *arXiv preprint arXiv:1807.02264*, 2018.

[30] Valentina Martina, Michele Garetto, and Emilio Leonardi. A unified approach to the performance analysis of caching systems. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 2040–2048. IEEE, 2014.

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

[32] Chris Mellor. Enterprise SSDs cost ten times more than nearline disk drives. https://web.archive.org/web/20221004225419/https://blocksandfiles.com/2020/08/24/10x-enterprise-ssd-price-premium-over-nearline-disk-drives/, 2020. Accessed: 2022-10-04.

[33] Newegg. Newegg: Dell intel d3-s4620 960gb sata 6gb/s 2.5-inch enterprise ssd. https://web.archive.org/web/20230921032102/https://www.newegg.com/dell-d3-s4620-960gb/p/2U3-000S-00104?Item=9SIA994K4B2373. Accessed: 2023-09-20.

[34] Newegg. Newegg: Seagate exos x18 st10000nm018g 10tb 7200 rpm 256mb cache sata 6.0gb/s 3.5" hard drives. https://web.archive.org/web/20230921032117/https://www.newegg.com/seagate-exos-x18-st10000nm018g-10tb/p/N82E16822185024?Item=N82E16822185024. Accessed: 2023-09-20.

[35] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. Facebook's Tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231, 2021.

[36] Timothy Pritchett and Mithuna Thottethodi. SieveStore: a highly-selective, ensemble-level disk cache for cost-performance. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 163–174, 2010.

[37] Liana V Rodriguez, Farzana Beente Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *FAST*, pages 341–354, 2021.

[38] Shan-Hsiang Shen and Aditya Akella. An information-aware qoe-centric mobile video cache. In *Proceedings of the 19th annual international conference on Mobile computing & networking*, pages 401–412, 2013.

[39] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 413–425, 2019.

[40] Leon Sixt, Evan Zheran Liu, Marie Pellat, James Wexler, Milad Hashemi Been Kim, and Martin Maas. Analyzing a caching model. *arXiv preprint arXiv:2112.06989*, 2021.

[41] Zhenyu Song, Daniel S Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, et al. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, 2020.

[42] Zhenyu Song, Kevin Chen, Nikhil Sarda, Deniz Altınbüken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi. HALP: Heuristic aided learned preference eviction policy for YouTube content delivery network. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[43] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, Kai Li, Wen Xia, Yucheng Zhang, Yujuan Tan, Phaneendra Reddy, Leif Walsh, et al. RIPQ: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, 2015.

[44] Tianqi Tang, Sheng Li, Lifeng Nai, Norm Jouppi, and Yuan Xie. Neurometer: An integrated power, area, and timing modeling framework for machine learning accelerators industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 841–853. IEEE, 2021.

[45] 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.

[46] Olivier Verscheure, Chitra Venkatramani, Pascal Frossard, and Lisa Amini. Joint server scheduling and proxy caching for video delivery. *Computer Communications*, 25(4):413–423, 2002.

[47] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ml-based lecar. In *HotStorage*, pages 928–936, 2018.

[48] Hua Wang, Jiawei Zhang, Ping Huang, Xinbo Yi, Bin Cheng, and Ke Zhou. Cache what you need to cache: Reducing write traffic in cloud cache via "one-time-access-exclusion" policy. *ACM Transactions on Storage (TOS)*, 16(3):1–24, 2020.

[49] Qiuping Wang, Jinhong Li, Patrick PC Lee, Tao Ouyang, Chao Shi, and Lilong Huang. Separating data via block invalidation time inference for write amplification reduction in Log-Structured storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 429–444, 2022.

[50] Nan Wu and Pengcheng Li. Phoebe: Reuse-aware online caching with reinforcement learning for emerging storage models. *arXiv preprint arXiv:2011.07160*, 2020.

[51] Gang Yan and Jian Li. RL-Bélády: A unified learning framework for content caching. In *Proceedings of the 28th ACM International Conference on Multimedia*, pages 1009–1017, 2020.

[52] Dongsheng Yang, Daniel S Berger, Kai Li, and Wyatt Lloyd. A learned cache eviction framework with minimal overhead. *arXiv preprint arXiv:2301.11886*, 2023.

[53] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. Fifo queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 130–149, 2023.

[54] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. Leaper: A learned prefetcher for cache invalidation in lsm-tree based storage engines. *Proceedings of the VLDB Endowment*, 13(12):1976–1989, 2020.

[55] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission optimization for Google datacenter flash caches. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1021–1036, 2022.

[56] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, Homer Wolfmeister, and Junaid Khalid. Cachesack: Theory and experience of google's admission optimization for datacenter flash caches. *ACM Transactions on Storage*, 19(2):1–24, 2023.

[57] Jieming Yin, Subhash Sethumurugan, Yasuko Eckert, Chintan Patel, Alan Smith, Eric Morton, Mark Oskin, Natalie Enright Jerger, and Gabriel H Loh. Experiences with ML-driven design: A NoC case study. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 637–648. IEEE, 2020.

[58] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. OSCA: An Online-Model based cache allocation scheme in cloud block storage systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 785–798, 2020.

[59] Yu Zhang, Ke Zhou, Ping Huang, Hua Wang, Jianying Hu, Yangtao Wang, Yongguang Ji, and Bin Cheng. A machine learning based write policy for SSD cache in cloud block storage. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1279–1282. IEEE, 2020.

[60] Mark Zhao, Satadru Pan, Niket Agarwal, Zhaoduo Wen, David Xu, Anand Natarajan, Pavan Kumar, Shiva Shankar P, Ritesh Tijoriwala, Karan Asher, Hao Wu, Aarti Basant, Daniel Ford, Delia David, Nezih Yigitbasi, Pratap Singh, Carole-Jean Wu, and Christos Kozyrakis. Tectonic-Shift: A composite storage fabric for large-scale ML training. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023.

[61] Giulio Zhou and Martin Maas. Learning on distributed traces for data center storage systems. *Proceedings of Machine Learning and Systems*, 3:350–364, 2021.

[62] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. Improving cache performance for large-scale photo stores via heuristic prefetching scheme. *IEEE Transactions on Parallel and Distributed Systems*, 30(9):2033–2045, 2019.

9 Artifact Appendix

Abstract

Our artifact is targeted at those seeking to reproduce the results found in the Baleen paper. It contains a Python simulator, an implementation of our cache residency model (*episodes*), and scripts for downloading traces. You may view our artifact and its README at `https://github.com/wonglkd/Baleen-FAST24`.

Scope

Our artifact allows users to easily 1) test our Python cache simulator with small-scale experiments, and 2) plot paper figures using supplied intermediate results.

We list the specific key claims and corresponding figures:

1. Baleen reduces estimated total cost of ownership (TCO), peak backend load (Fig 1) and miss rates (Fig A.1) (`fig-01a,08,13-202309-tco.ipynb`, `fig-01bc,17-202309.ipynb`)
2. Baleen does so across a range of traces (Fig 8, Fig 9) (`fig-01a,08,13-202309-tco.ipynb`, `fig-09-202309.ipynb`)
3. Baleen performs well across a range of cache sizes and flash write rates (Fig 10, 24, 25) (`fig-10a,24-wr-20230414.ipynb`, `fig-10b,25-csize-20230424.ipynb`)
4. Baleen benefits from smart prefetching that predicts the right range (Fig 13) and when to prefetch (Fig 14) (`fig-13,14-prefetching-20230424.ipynb`)

We also include additional notebooks that:

1. show how Baleen-TCO picks the optimal write rate (Fig 12), (`fig-01a,08,13-202309-tco.ipynb`)
2. show breakdown of benefit at peak (Fig 18) and (`fig-18-peak-hrs-20230424.ipynb`)
3. describe statistical properties of the workloads (Fig 7). (`fig-07,19,20-tracestats-20230504.ipynb`)

Caveats

When reproducing the results, we expect trends to be the same but small differences in the actual results due to two reasons: 1) Meta's exact constants for the disk-head time function will not be released, meaning that results will not be exactly the same; instead, in the released code, we use constants (seek time and bandwidth) measured on the hard disks in our university testbed; 2) the testbed code modified a proprietary internal version of CacheLib and that will not be released at this time. However, we expect the simulator to closely match the testbed (and have presented supporting evidence to that effect).

While all the necessary code and data is supplied to reproduce our results, setting up the simulator with a cluster scheduling system would be recommended if re-running all experiments (624 machine-days were utilized; each simulation of a ML policy takes at least 30 minutes, multiplied by 7 traces and 10 samples each). Helper code is included to facilitate runs on a cluster, but this will need to be adapted for your own cluster (see `BCacheSim/episodic_analysis/local_cluster.py`).

Contents

Our artifact includes the full traces used in the paper, a Python module (BCacheSim) that contains the flash cache simulator, an implementation of the episodes model, and code to train the policies. Further detail on the directory structure can be found in the README.

We also provide a walkthrough video that shows the authors reproducing the results on the Chameleon Cloud platform: `http://tiny.cc/BaleenArtifactYT`

Hosting

The artifact is hosted in a GitHub repository, in the main branch: `https://github.com/wonglkd/Baleen-FAST24`. For ease of reproduction, the artifact is also hosted on the Chameleon Cloud platform, a free academic cloud supported by NSF: `https://www.chameleoncloud.org/experiment/share/aa6fb454-6452-4fc8-994a-b028bfc3c82d` Users can choose to either use the artifact on their own machines or Chameleon.

Requirements

If using Chameleon Cloud, no local dependencies are required apart from the ability to SSH and a web browser. If using your own computer, the primary software dependency is Python 3.10 with specific packages listed in a `requirements.txt` in the repository. If you wish to run experiments in parallel on a cluster, a job scheduling system like Brooce (which we used) is recommended.

Baleen was developed on the Carnegie Mellon University Parallel Data Lab's Emulab testbed using Meta traces.

Time to reproduce

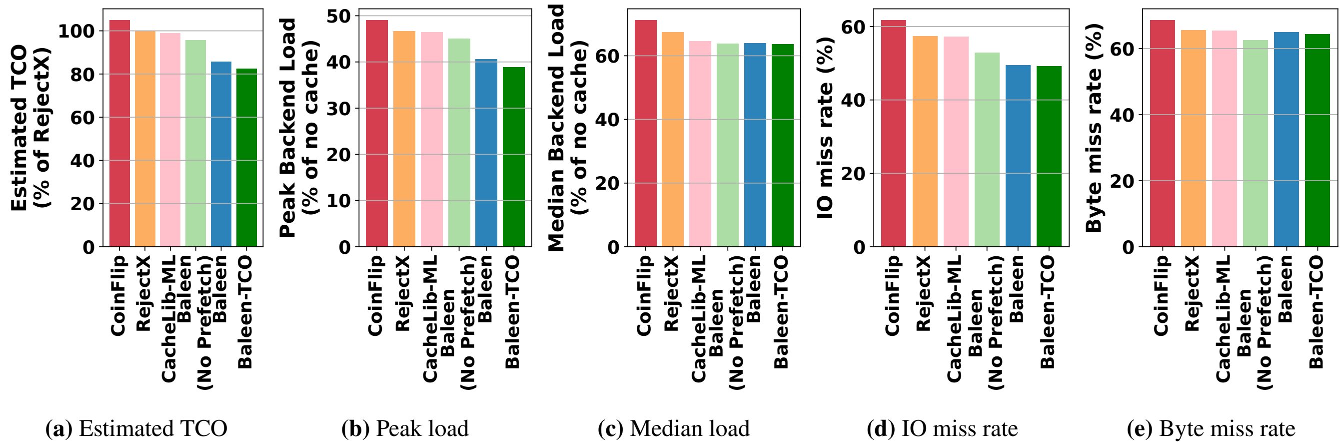
About 3 hours is required on Chameleon Cloud to run a set of basic experiments, and plot figures using the intermediate results supplied. To re-run all experiments from scratch would take 624 machine-days (based on the logged time it took to simulate the runs used). As a guideline, each simulation of a ML policy takes at least 30 minutes, multiplied by 7 traces and 10 samples each.

Troubleshooting and suppoort

A list of common issues and remedies is included in the README. GitHub issues are the preferred means of communication. You may also contact the corresponding author via email.

A Supplemental Material

A.1 Comparison to IO miss rate and bandwidth miss rate



(a) Estimated TCO **(b)** Peak load **(c)** Median load **(d)** IO miss rate **(e)** Byte miss rate

Figure 16: We provide IO miss rate and byte miss rate, two commonly used caching metrics, for comparison.

A.2 Median DT

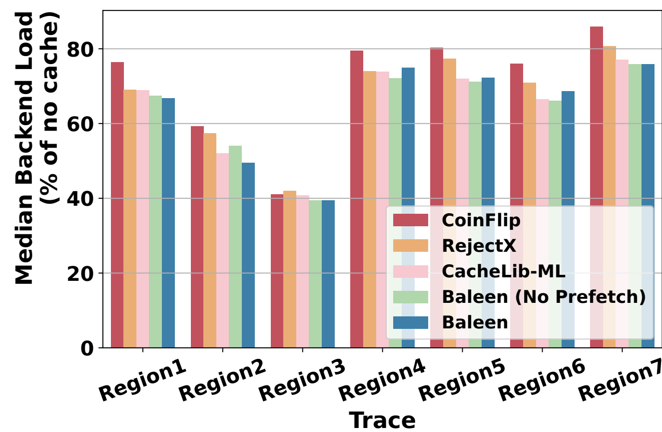
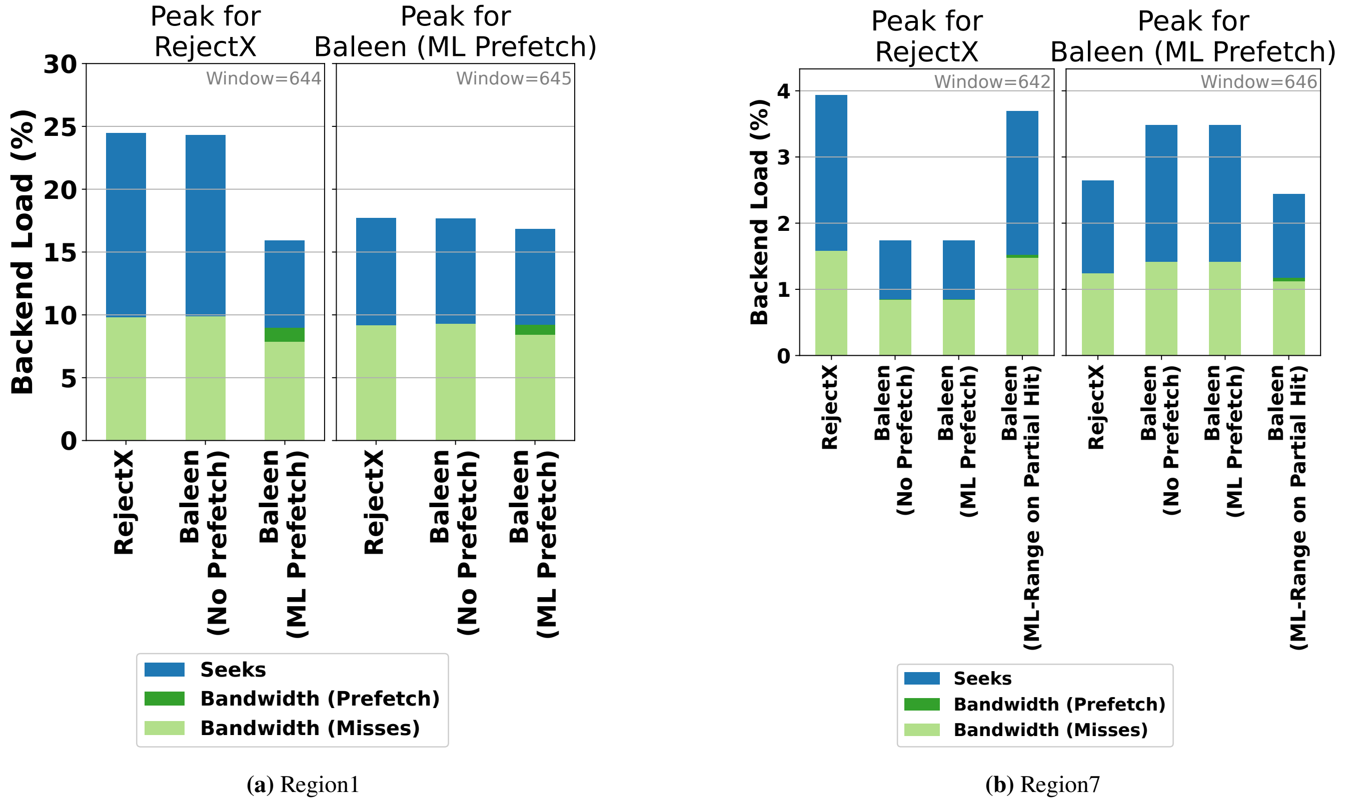


Figure 17: Median DT

A.3 Breakdown of DT during peak periods



(a) Region1

(b) Region7

Figure 18: Breakdowns of DT at Peaks. Each graph shows the peak 10-min window for that setup. Baleen's DT reduction is mostly due to reduced seeks.

In Fig 18a, we break down DT at the peak hours and show most of the Peak DT reduction is from eliminating seeks rather than data transfer. This is in line with how prefetching saves DT.

Fig 18 shows policies' performance at the respective peak windows for Baleen and RejectX. The peak window can differ from policy to policy, as one policy may be good at dealing with a traffic pattern that causes peaks for other policies, but be foiled by a pattern that is handled well by others. This makes optimizing the peak a whack-the-mole game. Baleen's worst time intervals are those in which prefetching is not beneficial. This suggests that a policy wanting to optimize Peak DT would be aware of the current load level and able to adapt to it.

A.4 TCO function: step-by-step

$$\mathrm{TCO}_1 \propto \frac{\mathrm{PeakDT}_1}{PeakDT_0} \cdot \#HDDs_0 + \frac{Cost_{SSD}}{Cost_{HDD}} \cdot \frac{\mathrm{FlashWR}_1}{FlashWR_0} \cdot \#SSDs_0 \tag{4a}$$

$$TCO_1 \propto PeakDT_1 \cdot R_1 + FlashWR_1 \cdot R_2 \tag{4b}$$

$$R_1 = \frac{1}{PeakDT_0} \tag{4c}$$

$$R_2 = \frac{1}{FlashWR_0} \cdot \frac{\#SSDs_0}{\#HDDs_0} \cdot \frac{Cost_{SSD}}{Cost_{HDD}} \tag{4d}$$

$$= \frac{1}{FlashWR_0} \cdot \frac{1}{36} \cdot \frac{170}{281} \tag{4e}$$

We calculate relative TCO savings using the Peak DT saved with our baseline admission policy RejectX ($PeakDT_0$), and relative to the default target flash write rate ($FlashWR_0$). From [35], we know that each node has 1x 1-TB SSD and 36x 10-TB HDDs ($\frac{\#HDDs_0}{\#SSDs_0} = 36$).

From [35], we know that each node has 1x 1-TB SSD and 36x 10-TB HDDs ($\frac{\#HDDs_0}{\#SSDs_0} = 36$). We substitute the 2023 price of a 10TB HDD ($281) and a 1 TB SSD ($170) on Newegg [33, 34] ($\frac{Cost_{SSD}}{Cost_{HDD}} = \frac{170}{281}$), i.e., the HDD is 6x cheaper per TB than the SSD. (For comparison, a 2020 industry report showed a 10x difference [32].)

A.5 Comparison to Flashield

We compared Baleen to Flashield, a state-of-the-art ML baseline. We adapted the implementation of Flashield used in the S3-FIFO paper in SOSP 2023 [53]. Flashield was worse than our RejectX baseline.

In practice, we found that a disadvantage of this approach is that DRAM lifetimes are too short to yield useful features. (Flashield assumes a 1:7 DRAM:Flash ratio, whereas Tectonic has a 1:40 ratio.)

Flashield failed on half the trace samples due to insufficient training data, because it relies on items' hits in DRAM for its features and labels. With DRAM lifetimes of seconds-to-minutes, most items never receive DRAM hits. Considering only workloads favorable to Flashield (that it could train a model on), Baleen outperformed Flashield by 18%.

A.6 Comparison to CacheLib ML

CacheLib ML is a ML model that Meta used in production for 3 years, which was first described by Berg *et al* [5]. Baleen uses the same ML architecture (GBT) and serving (inference) setup, but a different training setup (episodes and optimizing DT instead of hit rate). Based on this, we assert that Baleen's architecture is feasible for production with acceptable inference overhead. Meta's implementation is proprietary but general lessons learnt from it were described in §6.

A.7 Comparison to LRB's Relaxed Belady

LRB [41] introduces Relaxed Bélády for eviction, which only considers objects for eviction beyond a time it calls the Belady boundary. Like our OPT's use of the assumed eviction age, it prunes the decision space making it more efficient; our OPT is able to make stronger assumptions (due to the flash admission context), and train ML at a higher granularity of disjoint episodes, whereas LRB still operates at the finer granularity of accesses and is choosing which object is *more likely* to be good (has higher Good Decision Ratio) whereas OPT can determine which object is better to admit).

A.8 Workloads

The Region1 and Region2 traces were recorded from different clusters over the same 7 days in Oct 2019, while the Region3 trace was recorded from another cluster over 3 days in Sep 2019. Region4 was recorded over 7 days in Oct 2021, and the remaining traces (Region5, Region6, Region7) were collected in Mar 2023.

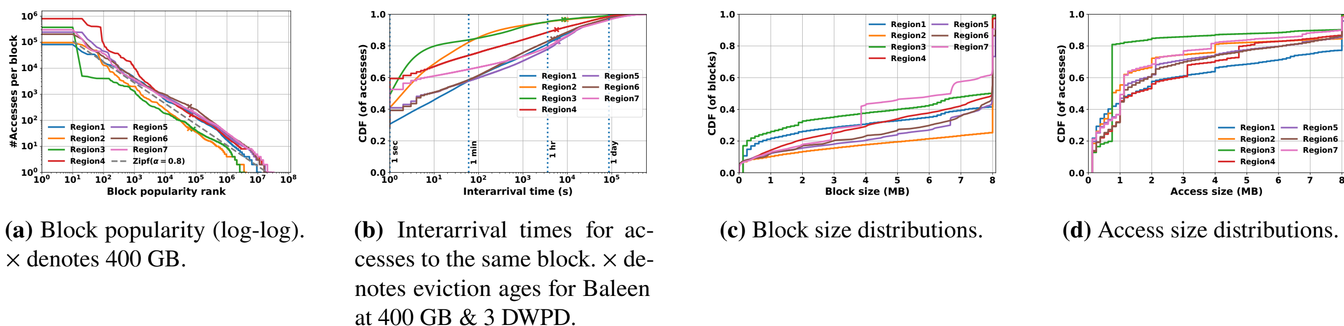
1. Regions 1-3 (2019): each a data warehouse
2. Region4 (2021): data warehouse
3. Region5 (2023): 10 "tenants", largest being data warehouse and blob store
4. Region6 (2023): 10 "tenants", largest being data warehouse and blob store
5. Regions 4-6 are from different geographical regions.

Each tenant supports 100s of applications. Data warehouse is storage for data analytics (e.g., Presto, Spark, AI training), with larger reads than blob storage. Blobs are immutable and opaque, and include media (photos, videos) and internal application data (e.g., core dumps). See the Tectonic [35] paper for further details.

Table 2: Full statistics of traces.

Dataset	**Year**	**Request Rate (s^{-1})**	**Avg Block Size (MB)**	**Access size (MB)**	**Compulsory miss rate[1]**	**One-hit-wonder rate[2]**	**PUT-Only Blocks**	**#PUT / #Acc**	**Admit-All Write Rate**
Region1	2019	244	5.70	3.41	18%	54%	46%	13%	316 MB/s
Region2	2019	106	5.07	2.85	39%	83%	81%	14%	121 MB/s
Region3	2019	139	6.71	2.42	19%	48%	46%	16%	45 MB/s
Region4	2021	406	5.87	2.87	14%	53%	40%	10%	280 MB/s
Region5	2023	364	6.84	2.62	18%	59%	33%	9%	480 MB/s
Region6	2023	404	6.77	2.74	14%	55%	38%	10%	478 MB/s
Region7	2023	426	5.71	2.23	17%	62%	38%	12%	492 MB/s

[1] Compulsory miss rate refers to the ratio of blocks to accesses;
[2] One-hit-wonder rate is the fraction of blocks with no reuse.



(a) Block popularity (log-log). × denotes 400 GB.

(b) Interarrival times for accesses to the same block. × denotes eviction ages for Baleen at 400 GB & 3 DWPD.

(c) Block size distributions.

(d) Access size distributions.

Figure 19: Distributions of block popularity, access interarrival times, block sizes, and access sizes for three traces. In a, lower values of α indicate it is harder to cache.

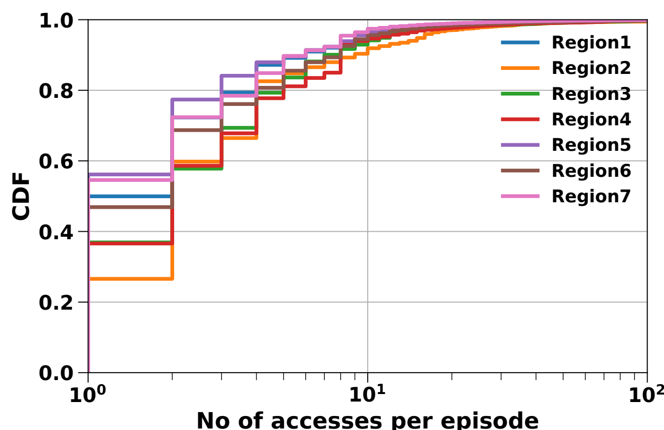


Figure 20: Distribution of hits per episode. This reflects the possible hits accrued from admitting an item.

A.9 Testbed hardware

As we did not have direct access to production hardware, we ran simulations (using our Python simulator) and testbed evaluations (using our modified version of CacheLib) on our academic testbed. This research testbed was a 24-node cluster, where each node has a 16-core Intel Xeon E5-2698 CPU, 64 GB of DRAM, Intel P3600 400 GB NVMe SSD, Seagate ST4000NM 4 TB HDDs, and runs Ubuntu 18.04. The SSDs and HDDs used are enterprise-grade. The size of the cluster does not affect the veracity of the testbed as each individual experiment run only involves one node; multiple nodes are used to speed up the completion of the experiments, as the total number of runs required is the total number of policy configurations multiplied by 7 traces and 10 samples from each trace.

A.10 Validating simulator and testbed

Fig 21 shows that testbed and simulator are faithful to production counters. We compare production counters for one day (collected on a per-minute basis and aggregated to 10-min intervals) to simulator and testbed results for a trace collected on the same day.

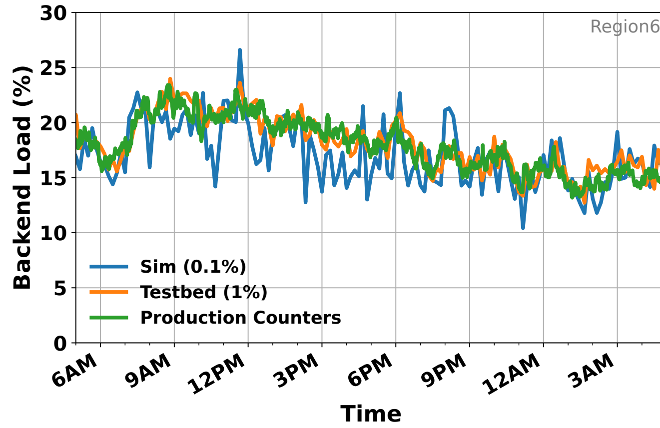


Figure 21: Sim-Testbed-Production comparison, RejectX, 1 day

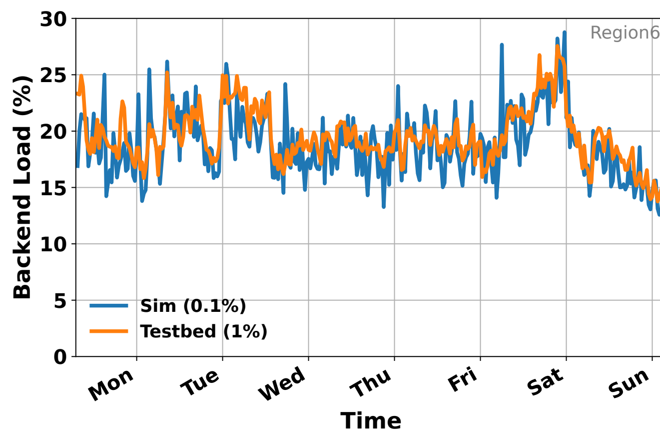


Figure 22: Testbed-Production comparison, Baleen, 1 week

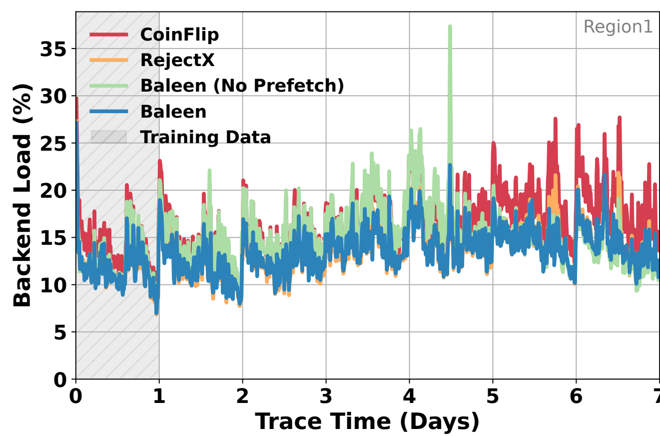


Figure 23: Testbed backend load over time, on the Region1 trace. Peak-to-mean ratio is 2. Granularity is 10 mins.

A.11 Write Rates and Cache Sizes for all traces

In Fig 10 we showed an average across all traces and selected traces; here we show data for all 7 traces.

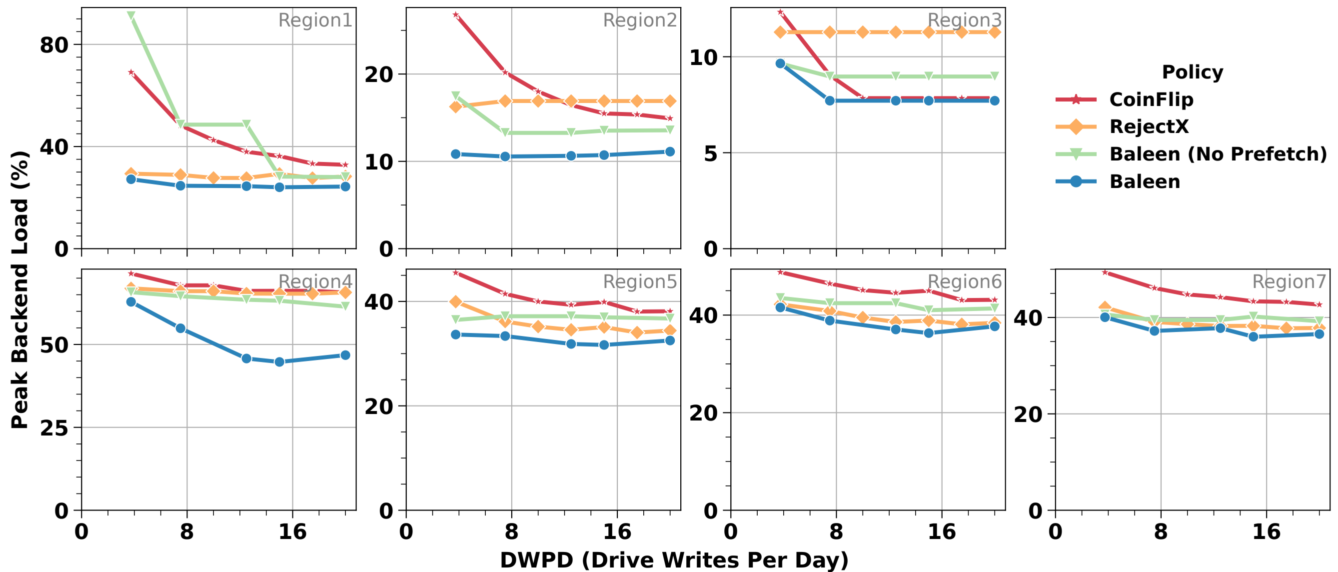


Figure 24: Benefits consistent as write rate increases.

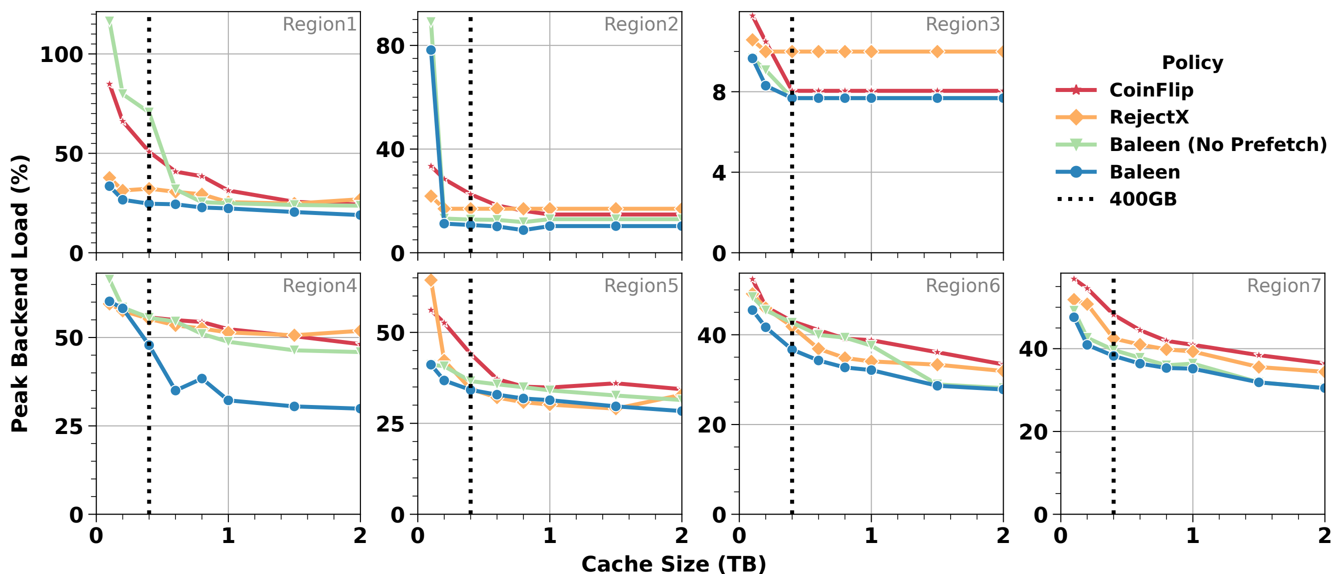


Figure 25: Benefits consistent as cache size increases.

B CacheLib deployment

B.1 CacheLib settings

CacheLib employs a region-based LRU, with different regions for different sizes. Since segments are uniformly 128 KB, we set region size to 142 KB to contain one segment each plus overhead.

We added functionality to CacheBench (CacheLib's benchmark suite) to replay Tectonic traces.

C Cache Transformer

GBMs are relatively simple and thus we also implemented more complex ML models for learning cache access patterns. Specifically, we add two deep models used to learn sequences in natural language processing:

Baseline: MLP feedforward A basic multilayer perceptron (MLP) feedforward model that takes the same features as our GBM model, i.e., only features from the current access, with a single hidden layer of size 80.

Cache Transformer architecture A Transformer [45] encoder that uses features from the prior h ($h = 16$) accesses in addition to the current access. Instead of sequences of words, it uses sequences of accesses. Further details are in Supp C.1.

We found that GBM performs best (0.2% better than Cache Transformer), despite only having features for the current access. This was contrary to our hypothesis that more historical information and access to the pattern of accesses would help model performance. Although we cannot dismiss the possibility that the Cache Transformer model is held back by our training process, a challenge we struggled with was the highly imbalanced classes. GBMs are known to be robust and work out of the box on many datasets. We observe that GBM produces the highest F1-score, i.e., it balances recall and precision the best. The MLP has the highest precision at the expense of recall. Baleen hence uses GBM given that it performs best and is the most efficient of the options explored.

C.1 Architecture

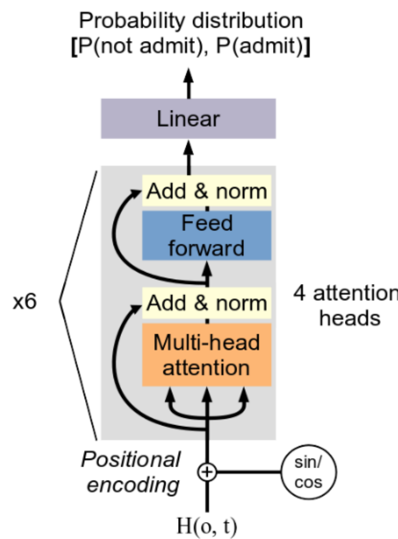


Figure 26: Cache Transformer architecture.

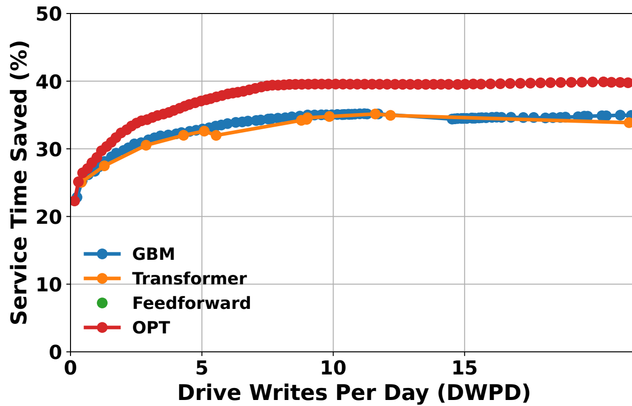
As shown in Fig 26, the Cache Transformer architecture consists of a series of Transformer encoders stacked together, with a linear classifier at the end. Before being passed to the first encoder, the windows are normalized and a sinusoidal positional encoding is applied. The encoders serve the purpose of learning and evaluating the self-attention between different accesses in the window. After the windows are passed through all the encoders, a final linear layer maps the last encoder's output to the model's prediction, which is represented as a probability distribution.

In summary: first, the model passes the sequence through a sinusoidal positional encoding to inject relative position information. Then, the encoded sequence is passed through 6 encoders with 4 attention heads each, followed by a linear layer that maps to a similar binary probability distribution to the MLP feedforward model.

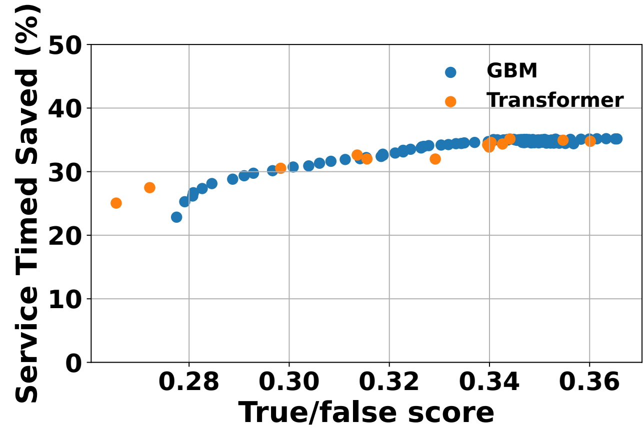
C.2 Training setup

Neural network models such as the Transformer used PyTorch for training and prediction. When training the Transformer neural network models, positive training examples are upsampled to balance out the classes and reduce the tendency to overfit. The MLP used for comparison had one 80-size hidden layer. Neural network training was done using RaySGD on a cluster with 8 Nvidia GeForce Titan X GPUs.

C.3 Evaluation



(a) Disk-head Time Saved against Write Rate.



(b) Disk-head Time Saved against true/false linear regression score.

Figure 27: Different architectures for ML admission. GBM is the best non-OPT policy. A 10%-trace was used. Mean DT is reported here, relative to no cache.

Table 3: Performance of different models, online and offline. h denotes the number of past accesses used as input into the model. Write rate and IO hit rate are from online simulations.

Model (h, history)	Loss	Offline accuracy	Online accuracy	Write Rate	IO hit rate	Precision	Recall	F_1
MLP feedforward (h = 1)	0.41	90.2%	88.5%	28.1 MB/s	48.1% (-8.6%)	85.6%	35.5%	0.502
Transformer (h = 16)	0.18	92.6%	89.5%	42.9 MB/s	49.3% (-6.5%)	66.7%	50.7%	0.576
GBM (h = 1)	-	93.8%	91.1%	37.9 MB/s	49.4% (-6.3%)	76.8%	51.9%	0.619
OPT	-	100%	100%	30.4 MB/s	52.7%	100%	100%	1

Seraph: Towards Scalable and Efficient Fully-external Graph Computation via On-demand Processing

Tsun-Yu Yang, Yizou Chen, Yuhong Liang, and Ming-Chang Yang
The Chinese University of Hong Kong

Abstract

Fully-external graph computation systems exhibit optimal scalability by computing the ever-growing, large-scale graph with constant amount of memory on a single machine. In particular, they keep the entire massive graph data in storage and iteratively load parts of them into memory for computation. Nevertheless, despite the merit of optimal scalability, their unreasonably-low efficiency often makes them uncompetitive, and even unpractical, to the other types of graph computation systems. The key rationale is that most existing fully-external graph computation systems over-emphasize retrieving graph data from storage through sequential access. Although this principle achieves high storage bandwidth, it often causes reading excessive and irrelevant data, which can severely degrade their overall efficiency.

Therefore, this work presents Seraph, a fully-external graph computation system that achieves optimal Scalability while toward satisfactory Efficiency improvement. Particularly, inspired by the modern storage offering comparable sequential and random access speeds, Seraph adopts the principle of *on-demand processing* to access the necessary graph data for saving I/O while enjoying the decent speed in random access. On the basis of this principle, Seraph further devises three practical designs to bring excellent performance leap to fully-external graph computation: 1) the hybrid format to represent the graph data for striking a good balance between I/O amount and access locality, 2) the vertex passing to enable efficient vertex updates on top of hybrid format, and 3) the selective pre-computation to re-use the loaded data for I/O reduction. Our evaluations reveal that Seraph notably outperforms other state-of-the-art fully-external systems under all the evaluated billion-scale graphs and representative graph algorithms by up to two orders of magnitude.

1 Introduction

Graphs have been broadly used in many fields, such as networking [10], social media [6, 19, 45], and bioinformatics [16, 26], for their attractive structure to represent the entities as *vertices* and relations between entities as *edges*. In practice, a graph is represented by two data structures: *vertex data* (denoted as V) holding the attributes of vertices, and *edge data* (denoted as E) comprising the edge lists, each of which enumerates the destination vertices connected with the same source vertex. Typically, *graph computation* involves reading the edge data for neighboring vertices and updating the vertices' attributes from/to their neighbors' attributes.

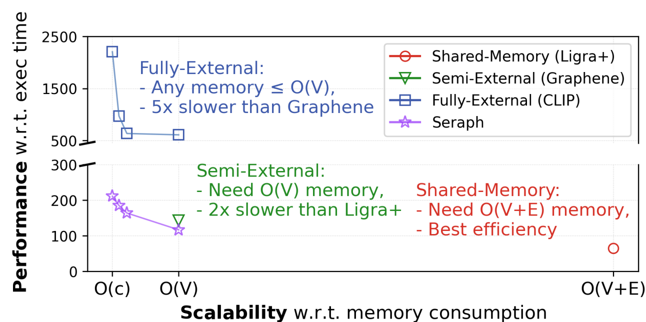


Figure 1: The performance-scalability spectrum among different state-of-the-art graph computation systems. The presented times are the average results of the evaluated graph systems with different amounts of memory described in §4.3.

Many single-machine *graph computation systems* have been developed to automate and optimize the process of graph computation, with the aim of high performance (i.e., low execution time). Recently, as graphs exponentially grow to have billions of vertices and edges, *scalability* is also essential for such systems. In this context, scalability refers to the capacity of a system to compute ever-growing, large-scale graphs within a single machine of common memory capacity. Therefore, how to design a scalable graph computation system that is also performant is the primary objective in this field.

In the following, we examine different kinds of single-machine graph systems from the aspects of scalability and performance. First, shared-memory graph systems (e.g., Ligra [36], Galois [32], Ligra+ [37]) require the entire graph data to be in memory (i.e., $O(V+E)$) for computing graphs with high performance. However, when targeting large-scale graphs, this approach is high-cost and difficult to scale as it

necessitates the machine with huge memory capacity.

To alleviate the issue of memory requirement, external-based graph computation systems, which can be further divided into *semi-external* and *fully-external* systems, are proposed to exploit the storage drives for graph computation at the cost of performance sacrifice. Semi-external graph computation systems (e.g., FlashGraph [47], Graphene [25]) are proposed to trade performance for alleviating the memory overhead by keeping the edge data in the massive-and-cheap storage while maintaining the vertex data in the small-and-expensive memory (i.e., $O(V)$). However, these systems only have limited scalability as their memory requirements still increase proportionally with graph sizes. Specifically, they cannot handle large-scale graphs with vertex data that exceeds the machine's memory capacity. Further, as graphs continue to grow to have billions of vertices, even keeping the vertex data in memory is not cost-effective [11, 17].

On the other hand, fully-external graph computation systems (e.g., GridGraph [50], CLIP [1], Lumos [42], V-Part [11]) further trade performance to offer the merit of optimal scalability; they can compute the large-scale graphs with a small amount of memory, which is independent from the graph data sizes. To accomplish this, they divide the large-scale graph into multiple subgraphs and keep them in storage; during runtime, each subgraph is iteratively handled so that the memory requirement can be effectively confined to computing only one subgraph. In other words, given c to be the available memory capacity of a machine, a fully-external graph computation system can use c to compute any size of large-scale graph by controlling the number of created subgraphs (i.e., $O(c)$). Thus, besides the edge I/O, fully-external systems require some additional I/Os to read/write the small-sized vertex data to establish each subgraph in memory by turns.

Fig. 1 summarizes the trade-off between scalability (i.e., memory consumption) and performance (i.e., execution time) among different kinds of state-of-the-art graph computation systems. Specifically, shared-memory-based Ligra+ demands a massive $O(V+E)$ memory for the highest performance. Semi-external-based Graphene requires $O(V)$ memory; compared to Ligra+, it needs 7.2x less memory yet exhibits 2x performance degradation. Fully-external-based CLIP demonstrates optimal scalability to compute the graph with any amount of memory that is smaller than or equal to $O(V)$; thus, it is able to use significantly less memory than the other types of systems. The minimum memory in Fig. 1 (i.e. $O(c)$) that CLIP uses is 17x less than Graphene and 127x less than Ligra+. However, we also observe a vast performance degradation: CLIP is significantly slower than Graphene regardless of the memory consumption; even given $O(V)$ memory, CLIP is around 5x slower than Graphene. In conclusion, although fully-external schemes offer the merit of optimal scalability, their severely-degraded performances often make them uncompetitive with other graph computation systems.

To fill this void, this work presents *Seraph*, a fully-external graph computation system that substantially boosts efficiency while offering optimal scalability.

To build such a system, we first recognize that most existing fully-external systems over-emphasize retrieving graph data from storage via sequential access. While this principle achieves high storage bandwidth, it also causes reading excessive-and-irrelevant data, particularly as many graph algorithms often exhibit sparse access patterns [15, 24, 29, 30]. Moreover, given that modern storage (e.g., solid-state drive (SSD)) offers comparable speeds for sequential and random access [38, 40], we seek the different principle of *on-demand processing* to access the necessary data with fine-grained I/O to save transferred data while exploiting the decent speed in random access. To investigate whether on-demand processing is promising for fully-external framework, we realize a baseline system to support on-demand processing and compare it against the state-of-the-art fully-external systems. Our evaluations, based on four types of storage devices, demonstrate the strong motivation that developing fully-external system with on-demand processing is a promising direction (see §2.3).

In light of this observation, we build Seraph upon *on-demand processing*. Moreover, we propose *three practical designs specially tailored for the framework of on-demand processing* to achieve further performance improvement. First, we observe that the traditional method for representing edge data has its pros and cons: it creates a good locality for accessing vertices yet increases the overhead of locating and reading edges. To this end, we present a new format, called *hybrid format*, to store the graph data by striking a good balance between locality for vertices and overhead for edges (see §3.2). Second, based on the hybrid format, we further propose *vertex passing* to enable efficient vertex updates by delaying and aggregating the vertex updates to the same subgraph via in-memory buffers (see §3.3). Third, although on-demand processing reads the necessary data, a common I/O block is typically way larger than an edge list. This mismatch inspires us with the opportunity of I/O re-using and the proposing of *selective pre-computation* to asynchronously compute the current and future vertices on the fly (see §3.4).

We implement Seraph in C++ and compare it against several state-of-the-art fully-external graph systems. Our evaluations, based on billion-scale graphs and representative graph algorithms, reveal that Seraph significantly outperforms the existing systems by up to two orders of magnitude. Further, with an increasing memory amount, Seraph also performs well and exhibits an up to 1.6x improvement over a recent semi-external graph system. Besides, we conduct investigation to justify each proposed design's effectiveness.

2 Background and Motivation

2.1 Background of Fully-external Systems

For large-scale graph computation on a single machine of limited memory capacity, fully-external graph computation systems necessarily divide the graph into multiple subgraphs

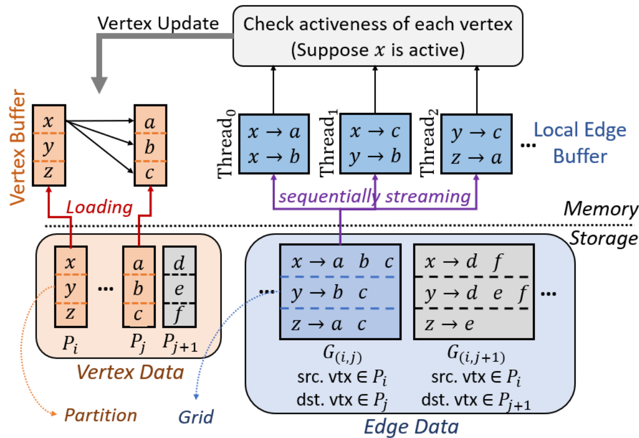


Figure 2: Typical system architecture of fully-external graph computation systems.

and handle them one at a time. In practice, as illustrated in Fig. 2, the vertices are generally divided into multiple disjoint *partitions*, and when processing the edges between two partitions (i.e., a subgraph), the memory requirement can be thus effectively limited to two vertex partitions involved (e.g., P_i and P_j) instead of the entire vertex data.

Based on the fundamental design described above, two common techniques are further adopted by the existing fully-external graph computation systems (such as [1,20,34,42,50]) for performance enhancement:

Grid Format for Edge Data. According to the results of vertex partitioning, existing fully-external graph computation systems typically store the edge data into *grid format*. In particular, a grid determines a set of edges by specifying one partition as source vertices and another (can be also the same one) as destination vertices. As shown in Fig. 2, the grid $G_{(i,j)}$ stores the set of edges whose source and destination vertices belong to partitions P_i and P_j, respectively. In other words, under the grid format, the edges of a source vertex can be split into multiple *segments* and are stored within different grids based on their destination vertices. As illustrated by Fig. 2, the edge list of the vertex x are split into two segments, which are (a, b, c) and (d, f), and are stored with grids $G_{(i,j)}$ and $G_{(i,j+1)}$, respectively.

Therefore, compared with storing the whole edge list of a vertex consecutively together (i.e., *row format* [28, 31]), processing the edge lists in grid format enjoys good locality of vertex access. This is because the required vertex attributes can be limited to only two partitions of vertices, which can entirely fit in the limited memory of fully-external graph computation systems. For instance, GridGraph [50] proposes 2D edge partitioning to split the edge data into smaller grids where the edges in the same grid share similar locality in accessing source and destination vertices.

Streaming-based Processing. On top of the grid format, existing fully-external graph systems generally apply *streaming-based processing*: they compute a graph by streaming the edge data from storage grid-by-grid. Given that all storage drives typically deliver high bandwidth with sequential access, streaming-based processing shows the generality of accommodating all types of storage.

As illustrated in Fig. 2, to compute a grid (e.g., $G_{(i,j)}$), the system first loads the two corresponding vertex partitions (e.g., P_i and P_j) into *vertex buffers* in memory. Next, as graph systems typically run in multi-threads for better performance, every thread sequentially streams edges from different parts of grid into its *local edge buffer* in memory. Following, the system identifies the vertices that need to be computed (i.e., *active vertices*), and then produces updates to their neighboring vertices based on the attributes of active vertices. For instance, GridGraph [50] checks the activeness of every grid by turns and sequentially streams the entire active grid from storage, in the granularity of 24 MB, to perform vertex update.

2.2 Existing Fully-external Systems

Many fully-external graph computation systems have been proposed for large-scale graph computation on a single machine. For example, GraphChi [20], which is the first fully-external graph computation system, divides a graph into multiple disjoint shards and sequentially load each shard into memory for computation. X-stream [34] proposes edge-centric processing model to stream and compute every edge for achieving high speed of sequential access. Inspired by X-stream, GridGraph [50] demonstrates a representative framework by splitting a graph with a smaller granularity (called a grid) to improve data locality; it achieves significant performance enhancement over GraphChi and X-stream.

Later, many systems propose optimization based on the framework of GridGraph. For instance, CLIP [1] introduces state-of-the-art optimization for streaming-based processing by asynchronously re-computing the loaded grid multiple times to increase data utilization and accelerate the convergence of graph algorithms. This feature makes CLIP specialized for *asynchronous graph algorithms* (e.g., vertex values following monotonicity [43]). Similar to CLIP, Lumos [42] also attempts to re-compute the loaded grid, but it aims to optimize *synchronous graph algorithms* which require synchronous semantics: a vertex can only observe the values from the last iteration. Thus, Lumos performs future computation on a vertex only if it receives all the updates from its neighbors. Due to this strict requirement, Lumos is more suitable for optimizing the algorithms which naturally demand synchronous semantics. Therefore, although both CLIP and Lumos improve over GridGraph with future computation, they are specialized for different sets of algorithms, respectively. On the other hand, Wonderland [46] applies the graph abstraction technique from visualization systems to streaming-based processing. However, the abstraction-guided processing only works for accelerating the convergence of path-based algorithms such as shortest path.

V-Part [11], a recent fully-external system, proposes a novel

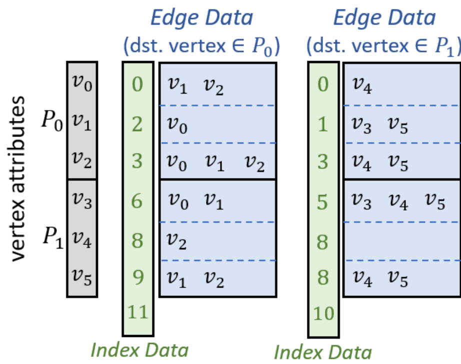


Figure 3: Data structure of GridGraph-ODP.

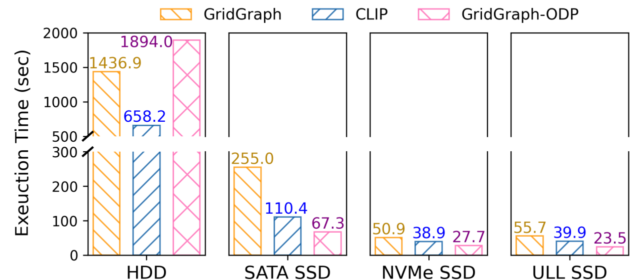


Figure 4: Evaluation of BFS on Twitter graph [9] with different storage devices.

framework different from GridGraph. V-Part stores the destination and source vertices into different structures. Each destination contains only attributes, while each source is associated with vertex ID, attribute, and offset pointing to its edge list, forming the source vertex table. To process a partition, V-Part first loads the destination vertices and then streams the source vertex table of the same partition into memory. Next, V-Part on-demand reads the active edge lists based on the offsets and updates the destination vertices. Finally, V-Part requires a stage, called mirror update, to synchronize the values/attributes between source and destination to facilitate vertex updates.

Nevertheless, V-Part fails to utilize on-demand processing thoroughly. Although V-Part on-demand accesses the edge data, it still streams the source vertex table from storage in a partition-based granularity, severely impacting its performance. Moreover, V-Part requires an extra overhead of mirror update, making its performance often worse than the other system adopting streaming-based processing (which will be shown in §4.1). Thus, to study which processing principle is more suitable for fully-external framework, we realize our own baseline system with on-demand processing and compare it against the state-of-the-art system with streaming-based processing in the next section.

2.3 Motivation: Streaming-based Processing versus On-demand Processing

Although streaming-based processing often loads excessive-and-irrelevant data as it streams an entire grid even if there are only a few active vertices/edges, it retains the advantage of achieving high storage bandwidth. Thus, many prior work have attempted to optimize streaming-based processing from various perspectives based on fully-external framework [1, 42, 44]. However, since modern storage provides comparable sequential and random access speeds, we attempt to look for the other principle, on-demand processing, to save I/O while exploiting the decent speed of random access. To study which principle is more suitable for fully-external framework, this section aims to compare the baseline of on-demand processing against the optimized streaming-based processing.

To realize a baseline system with on-demand processing under fully-external framework, we follow the traditional grid format to store edge data but replace streaming-based processing with the on-demand one. To this end, we revamp a representative, fully-external system GridGraph [50] to support on-demand processing (called GridGraph-ODP). Please note that we did not make CLIP support on-demand processing because the CLIP's re-computation only provides a coarse-grained, grid-based control, which contradicts the core concept of on-demand processing.

Compare to GridGraph, we add *index data* into GridGraph-ODP to record the offset of each segmented edge list, as shown in Fig. 3. These data enable on-demand processing for edge data because GridGraph-ODP can easily locate and read the required edge lists into memory. For example, suppose v is vertex and i is index, the edge list of v_s is located between i_s and i_{s+1}. Besides, since the vertex attributes are all sequentially stored based on their vertex IDs[1] in the file, it is easy to locate and access an attribute via its ID. Therefore, GridGraph-ODP can on-demand access both vertex and edge data.

We compare GridGraph-ODP against GridGraph [50] and CLIP [1]. In particular, GridGraph represents the baseline of streaming-based processing, and CLIP [1] stands for the state-of-the-art optimization for streaming-based processing. We use the famous breath-first search (BFS) [30] as a case study because it incurs both dense and sparse access patterns during computation [2]. We evaluate BFS on the three systems with the traditional hard-disk drive (HDD) as well as three different types of modern solid-state drives (SSD): SATA SSD [39], NVMe SSD [40], and ULL SSD [38]. The experiments are conducted in the same environment as that described in §4, and the number of threads is set to four, which is the same configuration as GridGraph [50].

Fig. 4 depicts the results. First, we can observe that HDD negatively impacts GridGraph-ODP. Compared with GridGraph, although GridGraph-ODP can save around 8.7x in loading edge data, its performance degrades by -24.1%. This result implies that the random I/O severely degrades the speed of HDD, and streaming-based processing is an effective principle to obtain high storage bandwidth. Moreover, CLIP improves GridGraph and GridGraph-ODP by 53.7% and 65.2%. The reason is that CLIP leverages streaming-based processing

[1] Each vertex is assigned with a distinct value to be its identity.

and further accelerates the convergence of graph algorithms by re-computing the loaded chunks multiple times. Thus, CLIP vastly outperforms the other two systems on HDD.

The results on modern SSDs show a different trend. Although both GridGraph and CLIP perform better on a faster drive, GridGraph-ODP improves more outstandingly because on-demand processing can save I/Os while enjoying the decent speed in random access. Moreover, we can observe an inspiring fact that, even if CLIP applies state-of-the-art optimization to streaming-based processing, GridGraph-ODP, which represents the baseline of on-demand processing, works remarkably better. Specifically, GridGraph-ODP outperforms GridGraph and CLIP by 59.0% and 36.3% on average. As a result, given that SSDs have prevailed nowadays, these experiments strongly motivate us that *advancing the development of fully-external system tailored for on-demand processing* is a promising direction.

3 Seraph

3.1 Overview

Motivated by §2.3, we realize that on-demand processing is a promising direction to build fully-external graph computation system. This inspires this work to develop a new fully-external graph computation system, Seraph, based on the principle of on-demand processing. Moreover, Seraph incorporates three main designs that are specially tailored for the framework of on-demand processing to pursue further performance improvement.

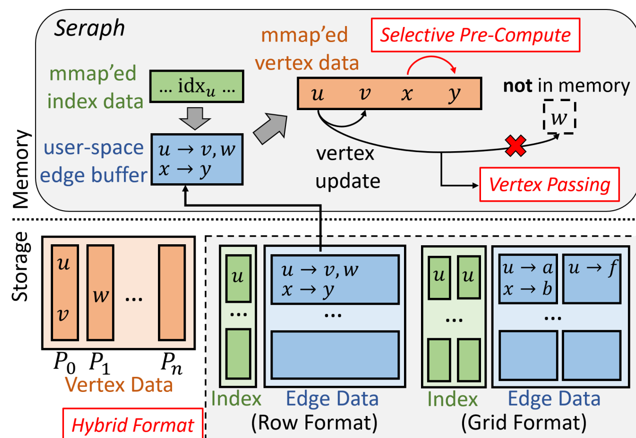


Figure 5: Architecture of Seraph.

Fig. 5 depicts Seraph's architecture, which combines the frameworks of fully-external graph computation and on-demand processing. To support fully-external framework, Seraph follows several traditional fundamentals such as keeping the entire graph data in storage and dividing the vertices into disjoint partitions. To reduce memory consumption, Seraph mmaps each vertex partition into memory and compute one partition at a time to create locality. To support on-demand processing framework, Seraph maintains index data to record the location of each edge list, just like our revamped GridGraph-ODP. Moreover, §3.2 presents a new *hybrid format* to split the edge data into both row and grid formats (e.g., for edge list of u, row format stores v and w, and grid format keeps a and f). It reduces I/O during graph computation by combining the advantages of both formats.

With these data structures, Seraph's execution flow is briefly illustrated as follows. It runs graph algorithms in iterations, and in each iteration, it first handles row format and then grid format. No matter which format is handled, Seraph computes one partition at a time. Specifically, graph computation involves identifying the active vertices, reading their edge lists, and updating the corresponding vertex attributes. However, when computing row format, the to-be-updated vertex attribute could not be inside memory (details in §3.3.1). To resolve this issue, §3.3 proposes *vertex passing* to delay the vertex updating for creating locality. On the other hand, §3.4 presents *selective pre-computation*. It explores the feature of asynchronous processing for further I/O reduction by re-using loaded data to compute the active vertices of future iterations in advance. Please note that the detailed execution flow involving the proposed designs will be elaborated in §3.5.

3.2 Hybrid Format

3.2.1 Observation

The motivation of hybrid format comes from the inefficiency of applying on-demand processing to the traditional grid format. Although it is common to utilize streaming-based processing with grid format, using on-demand processing with grid format is a double-edged sword. On one hand, grid format creates good locality of vertex access by confining the access range to two partitions only. On the other hand, it increases the overhead in reading edge lists and index data.

We first discuss how grid format negatively impacts the performance in reading edge lists. Compared with the edge list stored in row format, the grid format breaks an edge list into multiple segments and stores them in different grids, making on-demand processing issue a greater number of I/O blocks to read all the edge lists of the same source vertex in different grids. Under the example of Twitter graph[2] [9], the average number of 4KB pages to read the entire edge list (i.e., all neighbors) of a vertex is 3.88 in grid format, whereas row format only requires 1.03 pages, demonstrating a signifiant 73.5% improvement.

Next, reading the index data in grid format is inefficient due to the high ratio of redundant indexes. We call an index redundant if it points to an empty edge list, as such an index provides no information about the graph. Specifically, because the edge distribution of real-world graphs is typically skewed [12, 13], there is a high likelihood that a vertex has no edge list in a grid, leading to a high ratio of redundant indexes. In the same example of Twitter graph, over half (53.6%) of

[2]We assume the graph is divided into eight partitions.

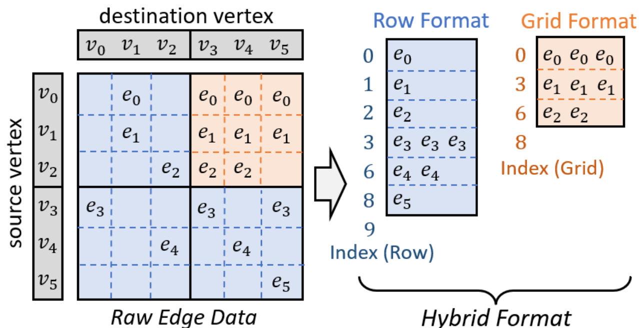


Figure 6: Example of hybrid format.

the indexes in grid format are redundant and provide no graph information. Further, as each grid maintains its own index data to enable on-demand processing, the total size of index data in grid format is considerably larger than that in row format. Thus, reading the index data in grid format becomes a non-negligible overhead to the overall performance.

Although row format seems better than grid format based on the above discussion, using row format in a fully-external system causes the issue of *accessing vertex attributes randomly*; the system may update arbitrary vertices in row format, rather than just a partition of vertices as in grid format. This leads to a serious issue that many vertex attributes that need to be updated may be on storage (i.e., not in memory). To update those attributes on storage, the system will extensively swap the pages between memory and storage, severely degrading the performance. Thus, §3.2.2 introduces *hybrid format* to resolve the dilemma between row and grid format.

3.2.2 Hybrid Format Construction

This section presents a new format, called *hybrid format*, to store the edge data while improving the performance of graph computation. The goal of hybrid format is to exploit the advantages of both row and grid formats. As shown in Fig. 6, it first stores the graph into grid format. Following, several grids are converted to row format to prevent the inefficiency in reading edge and index data, while retaining the rest in grid format to preserve good locality of vertex access.

To store the raw edge data into hybrid format, we first divide the edges into multiple chunks where every edge in the same chunk shares the same source and destination vertex partition(s). If a chunk contains many edges, computing the edges in the chunk will easily generate many vertex updates; storing the chunk in grid format is beneficial because we can create a good locality of vertex access during graph computation. On the contrary, for the chunks containing few edges, it is preferable to store them in row format for two reasons: (i) Because the edge lists in these chunks are small in general, storing them in row format can append all of the small edge lists together for reading them with the I/O block efficiently. (ii) Since the chunks of few edges tend to contain many empty edge lists, storing those chunks in row format prevents high ratio of redundant index data.

Hybrid format uses 8 bytes to store each index that points to the beginning location of each edge list; the edge list of a vertex v_s is recorded by s^{th} index and $(s+1)^{th}$ index. As shown in Fig. 6, row format contains $V+1$ indexes. Grid format requires $V/P+1$ indexes for each grid, so the total number of indexes in grid format is $(V/P+1)\times G$, where P is the number of partitions and G is the number of grids created (e.g., example in Fig. 6 is one grid in total).

To construct a graph into hybrid format, Seraph first sequentially reads the raw edge data[3], while deciding each edge chunk to be stored in grid or row format (details are discussed in §3.3.2). Next, based on the decision, it reads and re-distributes the raw edge data into grid or row format while creating the index data. Thus, the I/O complexity of constructing hybrid format is $O(5E+(V/P+1)\times G+V)$, whereas GridGraph takes $O(4E)$ to create grid format as it reads the raw edges and re-distributes them into different grids. Table 1 shows the construction times of GridGraph and Seraph (settings are described in §4). We can observe that the construction times of Seraph are roughly 30% slower than those of GridGraph. However, since graph construction is a one-time procedure per graph and can be performed offline, these costs are lightweight for both systems in terms of runtime performance, as shown in §4.1.

Table 1: Construction times of Seraph and GridGraph.

Time(sec)	Twitter	Gsh2015	Eu2015	RMAT
GridGraph	15.6	382.9	979.9	2285.5
Seraph	20.6	475.5	1294.5	2906.2

However, although the chunks that strongly require good locality of vertex access will be stored in grid format based on the above construction method, computing the edge lists in row format still results in several random accesses to vertex attributes. Therefore, §3.3 proposes a simple yet effective technique, called *vertex passing*, to tackle this issue. Please note that the further details such as the criterion for storing a chunk in the row or grid format will be discussed after introducing the concept of vertex passing.

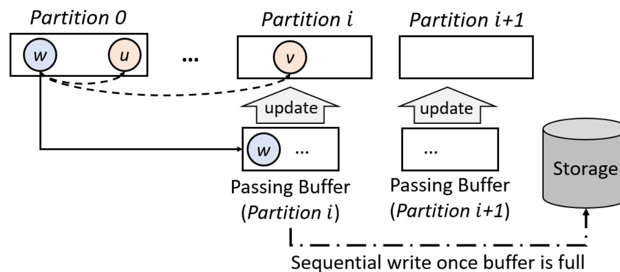


Figure 7: An example of execution on row format.

3.3 Vertex Passing

3.3.1 Design Concept

Vertex passing is proposed to resolve the issue of accessing vertex attributes randomly, as mentioned in §3.2. That is, it is

[3] Raw edge data and GridGraph present an edge with (`src`, `dst`). Hybrid format presents an edge with `dst` with the help of index data.

used when the to-be-updated vertex attributes are on storage. To enable vertex passing, each vertex partition is associated with an in-memory *passing buffer*. Suppose a vertex u is updated by its source vertex yet u is on storage, vertex passing will delay the update and transfer the information of the source vertex (i.e., active vertex) to the passing buffer of u's partition. Afterwards, once u is loaded into memory, vertex passing will merge the update in the passing buffer back to the attribute. Because a vertex partition shall be held in memory when Seraph currently computes it, every vertex can be held in memory by turns for update merging.

Fig. 7 shows an example about how vertex passing works. Suppose Seraph is computing P_0 (so the vertex attributes of P_0 shall be in memory), w is an active vertex, u and v are the neighbors of w. At this moment, the system can directly update vertex u but not vertex v since u is in memory and v is on storage; compelling the direct update to on-storage vertices will severely degrade the system's performance, as it involves extensive page swapping between memory and storage. Hence, vertex passing addresses this problem by delaying the on-storage update and transferring the update information of w to the passing buffer of P_i. Later, when Seraph is computing P_i, the vertex v can be loaded into memory and thus updated by the information stored in the P_i's passing buffer. In general, Seraph transfers the information of neighbor vertex ID and update value to the passing buffer, but the transferred information is configurable by the user based on the requirement of graph algorithms.

To mitigate the impact on memory usage in Seraph, each passing buffer is set to be negligibly small (e.g., 1 MB) compared to the vertex data size. Thus, each passing buffer is associated with a logging file to keep the vertex updates once the total size exceeds the in-memory passing buffer size. Seraph issues sequential I/O to write the information from the passing buffer to the corresponding file once the buffer is full. Similarly, sequential read is used for merging the information in the file back to vertex attributes. Please note that we use logging files to manage vertex updates instead of KV store because manipulating vertex updates is relatively simple; each update is only valid for one iteration, and they are bulky deleted after being used. Thus, we choose the straightforward implementation of logging files.

Although vertex passing requires I/O to transfer the updates, the overhead is generally small for three reasons. First, since vertex passing only transfers the update information of active vertices, it is effective for many graph algorithms activating a few vertices in most of the iterations. Second, as discussed in §3.2.2, a chunk stored in row format contains few edges; thus, even if all vertices are active and produce updates to their neighbors, the maximum overhead is bounded by the few edges inside each chunk. Third, because all transferred information is consecutively kept together in passing buffers and files, accessing them is efficient by using sequential I/O. Thus, vertex passing can efficiently tackle the issue of random vertex update caused by computation on row format.

The core concept of vertex passing (VP) is to delay operations as logs, which is a useful technique to create locality for different scenarios [21, 22]. For example, [22] exploits VP to optimize PageRank under shared-memory premise. It buffers all logs in DRAM efficiently and aims to improve VP's efficiency with more designs (e.g., lock-free layout). In contrast, our VP is naturally slower since the logs are recorded on storage via I/O. Thus, over-using our VP in fully-external environment will waste much I/O and eventually hurt Seraph's performance. In this regard, hybrid format and VP are proposed as a combination which complements each other with the aim of minimizing storage I/O.

3.3.2 Details in Hybrid Format Construction with Consideration of Vertex Passing

Seraph decides whether a chunk should be stored in row or grid formats by comparing their respective overheads. In particular, since this work targets fully-external environments (storage I/O is typically slower than CPU computation), each chunk is decided between two formats by considering their upper-bounded "I/O overheads" (more specifically, "I/O amounts"). That is, for a chunk C in grid format, the upper-bounded I/O amount includes reading all grid-related data (e.g., the chunk's vertex attributes, index data, and edge lists). For C in row format, the upper-bounded I/O amount is to log the updates generated by all edges in C. Therefore, the time complexity of gathering the above-mentioned information is linear to chunk size, while making decision is in constant time by simply comparing the two I/O-amounts.

On the other hand, a fully-external graph system typically reserves enough memory for holding two vertex partitions, one for the source partition and the other for the destination partition. In other words, when Seraph is computing a partition P in row format, it loads P into memory as source partition; there is an empty space for another partition to be destination partition. Thus, for every row of chunks (i.e., the chunks share the same source vertex partition), we can store one chunk in row format regardless of its number of edges because we can hold the chunk's destination partition in memory to absorb the vertex updates during graph computation. In fact, due to the natural locality of real-world graphs [49], many edges reside on the diagonal chunks (share the same source and destination partition). Thus, Seraph stores the diagonal chunks in row format by default.

Because different graph algorithms incur various access patterns, it is hard to tailor hybrid format for a specific access pattern. Thus, we construct hybrid format by heuristically assuming that all vertices are active for the following two reasons: 1) because the major bottleneck in graph computation is the dense access pattern under on-demand processing; optimizing dense access pattern is more beneficial than the sparse pattern in general, and 2) since the real-world graphs are typically skewed [12, 13], many chunks will be stored in

row format to enjoy the benefits even under the assumption of all active vertices. For example, we divides Twitter graph [9] into 64 chunks, and only 14 chunks are stored in grid format. For Eu2015 graph [7], all chunks are stored in row format as the vast majority of edges reside on the diagonal chunks.

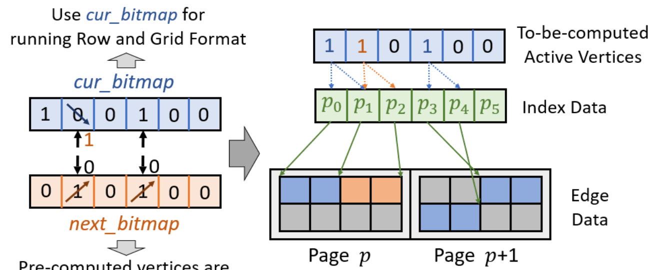


Figure 8: Example of pre-computation.

3.4 Selective Pre-Computation

Asynchronously computing the data of future iteration in advance is a well-known technique [1, 42]. As streaming-based processing often loads excessive-but-irrelevant data, existing work [1] exploits this concept to increase the utilization of the loaded grid by re-computing it many times. On the other hand, although Seraph leverages on-demand processing, there is still a granularity mismatch between a common I/O block size (e.g., 4096 bytes) and the typical size of an edge list (e.g., 132 bytes on average in Twitter). This mismatch provides the opportunity that the current and future active vertices[4] could reside on the same I/O block. Thus, *selective pre-computation* is introduced into Seraph to opportunistically re-use the loaded data for current active vertices to pre-compute the future ones asynchronously, reducing the total number of issued I/O.

However, under the framework of on-demand processing, recognizing whether the loaded data contain the information of future active vertices requires high implementation cost by traversing three different data structures (i.e., vertex attributes, indexes, and edges). To prevent this cost, we use vertex IDs for estimation due to the following three reasons. First, since all data structures are stored sequentially based on vertex ID, there is a high likelihood that two vertices reside on the same I/O blocks if their ID gap is small. Second, the implementation cost of tracking vertex IDs is low. Third, even for the worst case that we might spend a few extra I/O to load future active vertex, pre-computation merely beforehand performs computation that was supposed to happen in the next iteration, making it only require little cost yet offer the opportunity to re-use I/O. Based on our investigation, it is challenging to set an optimal value of ID gap for all scenarios because different algorithms/graphs have different features. Thus, we set the ID gap to be 32 by default as it is a reasonable value (by considering 4KB page and the average edge list size) and generally

[4]Current active vertices means the active vertices of current iteration. Future active vertices means the active vertices of next iteration.

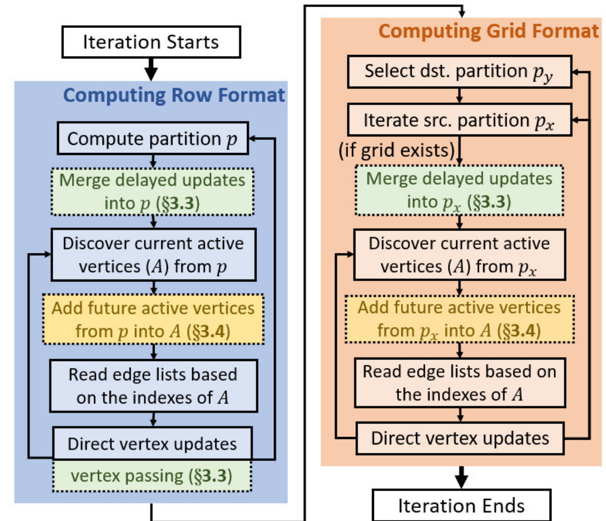


Figure 9: Execution flow in Seraph.

performs well for all graphs. In other words, Seraph will selectively pre-compute those future active vertices which are within the same gap based on the current active vertices to opportunistically re-use I/O.

Fig. 8 shows the mechanism of selective pre-computation. In particular, there are two bitmaps originally maintained in Seraph that record the activeness of each vertex: `cur_bitmap` and `next_bitmap` respectively indicate the active vertices of current and next iteration. In normal computation (i.e., without pre-computation), Seraph identifies and computes the active vertices based on the set-bits in `cur_bitmap`. In pre-computation, Seraph first determines the pre-computed vertices based on the method described above and move the set-bits from `next_bitmap` to `cur_bitmap`. In other words, all active vertices (including the current and future ones) are marked in `cur_bitmap`, and the pre-computed ones are removed from `next_bitmap` so that we will not compute them again in the next iteration. Finally, Seraph can simply perform graph computation based on `cur_bitmap`.

As Seraph stores graphs in hybrid format, it is essential to maintain consistency of the pre-computed vertices between row and grid formats to ensure the correctness of algorithms. Even when computing grid format, it is possible for a vertex to be pre-computed in one grid but not in another, as each grid is computed independently. To ensure consistency, Seraph first computes row format while modifying `cur_bitmap`. Next, Seraph uses the modified `cur_bitmap` to compute all grids to maintain consistency. However, pre-computation can only work when the graph algorithm generates future active vertices on the fly during graph computation. If this is not feasible for the algorithm, Seraph offers an option to disable pre-computation and run the algorithm normally.

3.5 Execution Flow

Based on the the available amount of memory specified by user, Seraph divides a graph into the number of partitions

that can hold two partitions of vertex data (i.e., source and destination partitions), one partition of index data, and other small data structures (e.g., edge buffer and active bitmap) in memory. With the information of partitions, Seraph constructs a graph into hybrid format based on the method described in §3.3.2. Next, Seraph performs graph computation based on the execution flow shown in Fig. 9. As mentioned in §3.4, Seraph computes row format first, and then computes the grid format, for each iteration. When computing a partition p, Seraph first merges all the delayed updates back to p to keep all attributes with the latest values. Next, each thread in Seraph will parallelly identify and record the set of active vertices in partition p; if pre-computation is enabled, future active vertices of the same partition will also be handled as mentioned in §3.4. Seraph converts the indexes of active vertices into page-aligned offsets. Later, Seraph exploits the kernel-level Linux Asynchronous IO (AIO) to group multiple required pages into one I/O request, and issue the request with direct I/O to read the edge data into local edge buffer. Finally, Seraph performs graph computation, and vertex passing described in §3.3 will be enabled when computing row format. Please note that, since Seraph uses memory mapping mechanism to reference the vertex and index data backed in files, Seraph accesses the needed vertices and indexes like accessing normal arrays, and does not declare extra user-space memory buffer for holding the vertex and index data.

Seraph computes row format by simply processing all partitions by turns, and grid format is computed by column-based execution order. In other words, when computing grid format, Seraph selects a destination partition p_y and then iteratively computes the source partition p_x if there is a grid between p_x and p_y. As p_y is fixed, all generated vertex updates can directly be absorbed in memory based on this column-based execution order. The computation in grid format ends when all the grids are computed by Seraph.

4 Evaluation

We implement Seraph in C++ and compare it against different state-of-the-art graph computation systems: fully-external (GridGraph [50], V-Part [11], CLIP [1], Lumos [42]), semi-external (Graphene [25]), and shared-memory (Ligra+ [37]) systems. As V-Part and CLIP are not open source, we implement their systems ourselves based on their papers.

We use breath-first search (BFS) [30], weakly connected component (WCC) [15], K-core (Kcore) [29, 35], all-pair shortest-path (APSP) [24, 41], and pagerank (PR) [14, 33] for evaluation. BFS is a typical algorithm for graph traversal by exploring the neighbors until all connected vertices are visited. WCC discovers the number of connected components of a graph; we implement WCC by the method of label propagation [48]. Kcore iteratively removes the vertices of degree less than k, and finally returns a subgraph where each vertex has the degree of at least k. APSP calculates the shortest paths from all vertices. Due to the high complexity of computing all the shortest paths, this evaluation leverages an approximate approach by randomly sampling 32 source vertices, and performs multi-source traversal from the sampled vertices [24, 41]. Finally, PR calculates the popularity of a vertex based on its neighbors' rank values. We run PR for four iterations and activate all vertices in each iteration. Please note that, except for PR constantly activating all vertices, the other evaluated algorithms activate a (dense or sparse) set of vertices in each iteration and represent various access patterns.

Table 2 lists the graphs used for evaluation in this work. The first three are billion-scale, real-world graphs from SNAP [23] and webgraph [3, 4]. In particular, Twitter [9] is social graph, while Gsh2015 [8] and Eu2015 [7] are web crawler graphs. Since the open-sourced graph datasets are all quite small, we use [18] to generate a large-scale graph (called RMAT) for testing the scalability. RMAT contains 8.6 billions of vertices and 112 billions of edges. Notably, because 4 bytes (`unsigned int`) is not enough to represent all the vertex IDs of RMAT graph, we use 8 bytes (`long`) to store the vertex ID for this graph in all systems instead.

Table 2: Evaluated graph datasets.

Graph Name	Num Vertices	Num Edges	Graph Size[5]
Twitter	42 M	1.4 B	11.2 GB
Gsh2015	988 M	33.88 B	271 GB
Eu2015	1.1 B	91.8 B	734 GB
RMAT	8.6 B	112 B	1.7 TB

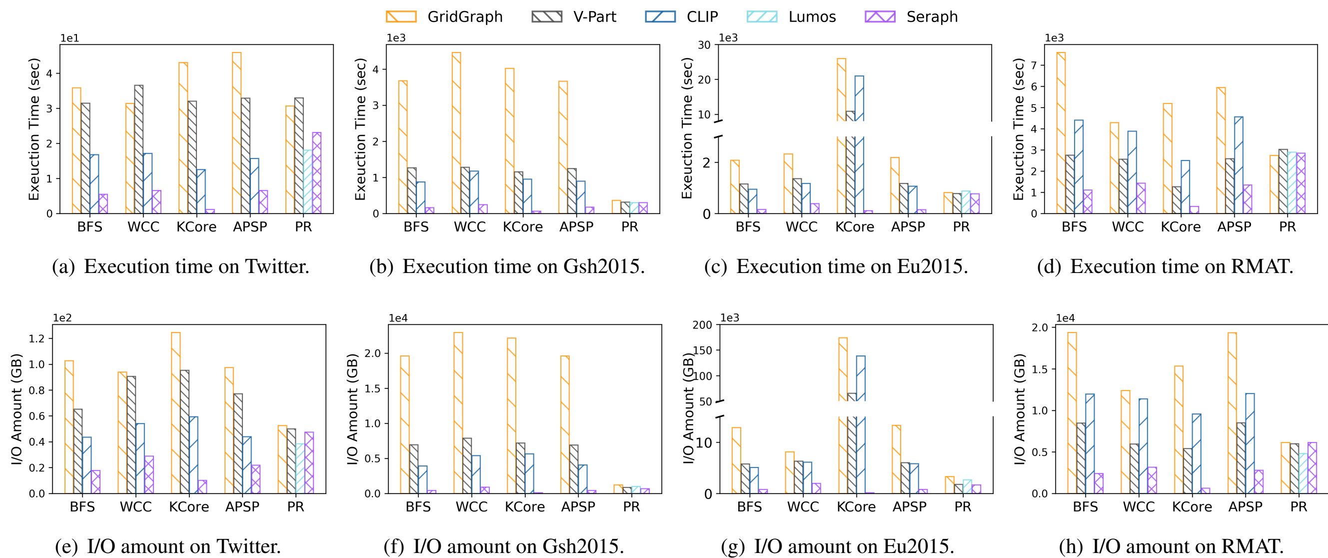
Table 3: Fully-external memory usages for §4.1 and §4.2.

Graph Name	Twitter	Gsh2015	Eu2015	RMAT
Memory Usage	130 MB	2.4 GB	2.7 GB	18 GB

To investigate fully-external graph systems, §4.1 and §4.2 conduct detailed comparisons and study the proposed design choices by offering each system with fixed amounts of memory that are adjusted based on the graph sizes. Specifically, we aim to compute the largest evaluated graph, RMAT, with a reasonable resource available for most people nowadays. Thus, each system is provided with 18 GB to compute RMAT, while the memory amounts for computing other evaluated graphs are also based on a similar ratio of each graph size. The exact memory usage for each evaluated graph are reported in Table 3. §4.3 further enhances the evaluation by examining each fully-external system with different memory amounts. Moreover, we also evaluate semi-external and shared-memory systems in §4.3 to have a comprehensive study about different types of single-machine graph systems.

To compare all types of graph systems on the same platform, all experiments are conducted on the same server: HPE ProLiant DL560 Gen10 server with Intel Xeon Platinum 8160 CPU and 32 x 32GB Dual Rank DDR4-2666 memory (1TB in total) on Debian GNU/Linux 9, and two 1TB Samsung NVMe SSD drives [40] with 6.0 GB/s sequential read bandwidth in total. We use `cgroup` to limit the available memory

[5]The size is measured by storing the graph in the format that each edge is represented by (source vertex ID, destination vertex ID).



(a) Execution time on Twitter. (b) Execution time on Gsh2015. (c) Execution time on Eu2015. (d) Execution time on RMAT.

(e) I/O amount on Twitter. (f) I/O amount on Gsh2015. (g) I/O amount on Eu2015. (h) I/O amount on RMAT.

Figure 10: Overall comparison among Seraph and other fully-external systems.

and `taskset` to confine the used cores. The number of threads for all systems is set to 16 because it is reasonable for commodity PCs nowadays.

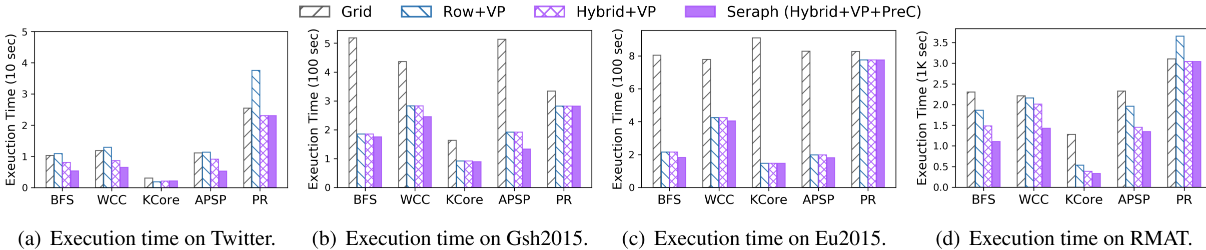
4.1 Fully-external Systems Comparison

This section compares Seraph against four state-of-the-art fully-external graph systems, which are GridGraph [50], V-Part [11], CLIP [1], and Lumos [42]. Fig. 10 illustrates the execution times (seconds) and I/O amounts (GBs) of running the chosen algorithms on different graphs.

We first discuss BFS, WCC, Kcore, and APSP; they are asynchronous graph algorithms, which offers room for optimization via exploiting the algorithmic feature, as discussed in Section 2.2. Overall speaking, GridGraph performs the worst among all systems since it naïvely adopts streaming-based processing (SBP). V-Part issues on-demand I/O to read the edge data, so it averagely improves GridGraph by 37.3% in execution time and 45.1% in I/O amount. However, V-Part's performance is impacted by 1) loading the source vertex table in a partition-based granularity and 2) requiring the overhead of mirror update to handle vertex updates. Thus, compared to CLIP which is an advanced version of GridGraph for asynchronous algorithms, V-Part is slightly slower than CLIP by 5.7% in execution time on average. Last but not least, Seraph performs the best among all systems. Compared to V-Part, Seraph on-demand accesses both edge and vertex data. Compared to CLIP, Seraph not just relies on the effectiveness of on-demand processing but it also leverages pre-computation to optimize asynchronous algorithms. In summary, since Seraph can effectively reduce I/O, it achieves decent performance correspondingly. For execution time (resp., I/O amount), Seraph outperforms GridGraph, V-Part, and CLIP, by 8.9x, 4.9x, and 4.0x (resp., 8.5x, 5.0x, and 4.5x).

Moreover, we can observe that Seraph is especially efficient in computing Gsh2015 graph. Taking BFS as an example; Seraph improves CLIP by 5.0x on Gsh2015 and 3.1x on Twitter. The reason is that running BFS on Gsh2015 takes lots of iterations to traverse the entire graph, and only a few vertices are activated in most iterations. This feature damages the systems adopting SBP as they typically read plenty of data in each iteration. By contrast, since Seraph on-demand accesses the necessary data, the main overhead is the number of issued I/Os, not the number of iterations. Thus, Seraph works better on Gsh2015 than other systems. A similar observation can be found on Kcore which takes lots of iterations to complete but only activates a few vertices in most iterations, providing Seraph more advantages in saving I/O than other systems.

Following, we discuss synchronous algorithm (i.e., PR). Compared to asynchronous ones, PR has to obey the strict synchronous semantics, as discussed in Section 2.2. Moreover, because PR is computation-intensive and constantly activates all vertices, on-demand processing does not show advantages, and all systems perform similarly. Nevertheless, Seraph is still slightly better in I/O as the graph in hybrid format is more lightweight than the graph structures in other systems. On the other hand, Lumos, to save I/O for synchronous algorithm via future computation, is specialized by imposing several constraints during graph computation. However, due to the feature of computation-intensiveness, loading less I/O only brings minor benefit. Please note that we replace the bar of CLIP with Lumos for PR in Fig. 10. Compared to GridGraph, V-Part, and Lumos, Seraph saves the I/O amount by 22.7%, 8.5%, and 7.1%, and improves the time by 11.1%, 10.4%, and 0.6%. Besides, although Lumos optimizes PR via I/O reduction based on GridGraph, the improvement is minor because (1) PR is computation-intensive in our testing, and (2) several designs in Lumos fail to run in fully-external environment. Thus, Lumos improves GridGraph by 39.8% on Twitter, but barely any improvement on the other evaluated graphs.



(a) Execution time on Twitter. (b) Execution time on Gsh2015. (c) Execution time on Eu2015. (d) Execution time on RMAT.

Figure 11: Performance studies of different major designs in Seraph.

4.2 Design Choices

This section demonstrates the performance impact of major designs in Seraph. Specifically, §4.2.1 reveals that hybrid format with vertex passing performs the best than the other two formats. Next, based on the hybrid format with vertex passing, §4.2.2 further studies the performance of selective pre-computation. The configuration of the system and environment in this section is the same as that in §4.1.

4.2.1 Hybrid Format and Vertex Passing

Before showing the results, we discuss the necessity of combining hybrid/row format with vertex passing. Since vertex passing creates the imperative locality for vertex access, computing hybrid/row format without vertex passing will cause the serious problem of memory thrashing. Based on our investigation, it severely degrades the execution time by at least two orders of magnitude, making the system intolerably slow.

Thus, in the following experiments, we compare the performance of grid format (denoted as *Grid*), row format with vertex passing (denoted as *Row+VP*), and hybrid format with vertex passing (denoted as *Hybrid+VP*), respectively. To clearly observe the effects of different formats, we disable selective pre-computation, which will be discussed in §4.2.2. Please note that, the hybrid format stores all edges of Gsh2015 and Eu2015 into row format. Thus, the results of *Row+VP* and *Hybrid+VP* are identical for Gsh2015 and Eu2015.

As revealed in Fig. 11, *Hybrid+VP* generally performs the best. Particularly, *Hybrid+VP* outperforms *Grid* by 37.4% on average, while *Hybrid+VP* averagely improves *Row+VP* (for Twitter and RMAT graphs) by 18.1%. Such improvement is because hybrid format strikes a good balance between utilizing row and gird formats. Compared to *Row+VP*, *Hybrid+VP* alleviates the overhead of vertex passing by storing several dense edge blocks in grid format. Compared to the traditional *Grid*, *Hybrid+VP* improves the efficiency of reading indexes and edge data. Moreover, the comparison between *Hybrid+VP* and *Grid* further implies that naïvely applying on-demand processing into the traditional fully-external designs will lead to limited improvements; proposing techniques suitable for on-demand processing (e.g., hybrid format and vertex passing) is essential for bringing improvement. Finally, *Row+VP* is not necessarily better than *Grid* because, for certain graphs and algorithms, over-using VP will degrade the overall performance instead by wasting too much I/O in transferring vertex updates.

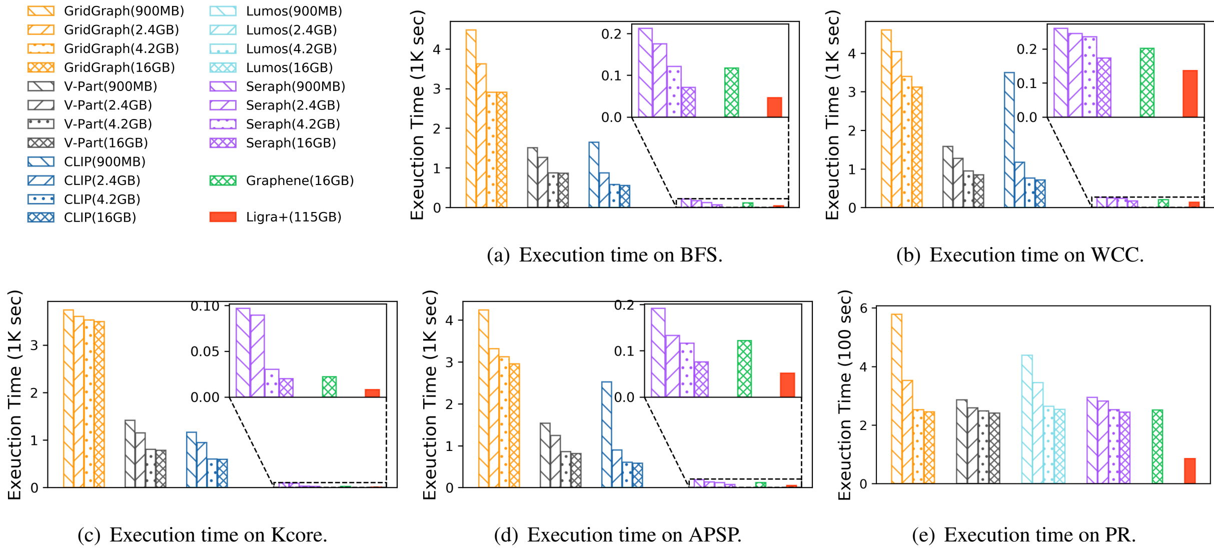
4.2.2 Selective Pre-computation

This section examines selective pre-computation based on the proposed hybrid format with vertex passing. The combination of designs is referred to as *Hybrid+VP+PreC*, and the results are presented in Fig. 11. Please note that, because the selective pre-computation is not feasible for PR, the result of *Hybrid+VP+PreC* is identical to *Hybrid+VP* in PR. We mainly discuss of the other four algorithms in the following. As a whole, selective pre-computation averagely improves *Hybrid+VP* by 16.1% in execution time. This improvement is achieved by re-using I/O opportunistically. Take the largest RMAT graph as example, pre-computation helps to improve execution time (resp., I/O amounts) by 25.6%, 29.1%, 14.5%, and 8.2% (resp., 25.8%, 25.1%, 15.2%, and 12.1%) in terms of BFS, WCC, KCore, and APSP. The results exhibit a similar trend between two metrics. Moreover, different graphs may lead to different amounts of improvement. Given that Twitter and RMAT have shorter diameter than Gsh2015 and Eu2015, the algorithms running on Twitter and RMAT typically incur denser access pattern than Gsh2015 and Eu2015. Thus, pre-computation has better improvement on Twitter and RMAT (21.5%) than Gsh2015 and Eu2015 (10.7%) because it can re-use more I/O under dense access pattern. A similar trends happens on KCore which usually has sparse access patterns. This makes pre-computation have a smaller impact on improving KCore (3.7%) compared to other algorithms (20.2%). However, pre-computation remains a reasonable design choice as it provides harmless benefit.

4.3 Evaluation with Different Amounts of Memory

This section evaluates with different memory amounts. Besides fully-external systems, we also include the state-of-the-art semi-external system (i.e., Graphene [25]) and shared-memory system (i.e., Ligra+ [37]). Particularly, Ligra+ leverages compression schemes to reduce runtime memory footprint. However, the compression program implemented in Ligra+ requires multiple in-memory edge-scale arrays, which significantly limits the graph scale that Ligra+ can handle. In fact, among all the evaluated graph datasets, the largest graph that Ligra+ can handle with our 1TB memory server is the Gsh2015 graph [8]. Hence, this section presents the results based on Gsh2015 graph.

For Gsh2015 graph, 16 GB is enough to hold the entire vertex and index data in memory (i.e., semi-external mode). Thus,



(a) Execution time on BFS.

(b) Execution time on WCC.

(c) Execution time on Kcore.

(d) Execution time on APSP.

(e) Execution time on PR.

Figure 12: Evaluation with different amounts of memory, which is adjusted based on the scale of Gsh2015 graph [8].

for fully-external systems, we vary the provided amounts of memory from 900 MB, 2.4 GB, 4.8 GB, to 16 GB. On the other hand, because Graphene has to run in semi-external mode and Ligra+ must hold the entire compressed graph in memory, we offer them 16 GB and 115 GB of memory to meet their requirements, respectively. It is worth nothing that the *memory requirements of both Graphene and Ligra+ proportionally increase with larger graph scales*. Taking RMAT as an example, Graphene and Ligra+ respectively require 132 GB and 1.7 TB (reported based on raw graph size), while Seraph can compute it with only 18 GB as shown in §4.1.

Fig. 12 shows that, for fully-external systems, their performances improve along with the increasing memory, and Seraph outperforms the other systems regardless of the provided amounts of memory. Moreover, Seraph shows a greater cost-effectiveness than other systems. Compared to fully-external ones, Seraph can use much less memory while achieving better performance. Compared to Graphene, Seraph(4.2GB) almost catches up the performance of Graphene(16GB), with only a minor 10.6% degradation on average. In semi-external mode, Seraph(16GB) averagely improves Graphene(16GB) by 1.31x due to the help of pre-computation. Finally, Ligra+ spends 7.2x more memory than Seraph to achieve an average 1.83x speedup, yet for certain algorithms like WCC, the speedup is only 1.27x. Ligra+ performs the best on PR (improves Seraph by 2.87x). This is because Ligra+ expensively keeps two versions of edge data (out-edges and in-edges) in memory and switches the access between them to resolve the computation-intensive issue of PR. Conversely, Seraph shows greater scalability by computing much larger graphs (e.g., Eu2015 and RMAT) that Ligra+ cannot handle.

5 Related Work

Besides fully-external graph systems, distributed graph systems [5, 13, 27, 49] also show high scalability in graph computation by splitting a graph across multiple machines. For example, Pregel [27] is the first distributed system proposing vertex-centric programming model. Based on vertex-centric model, PowerGraph [13] proposes to optimize the graph computation on natural graphs. Gemini [49] adopts many optimizations to greatly improve the efficiency. Although distributed systems also demonstrate the capability of large-scale graph computation, these approaches are high-cost as user need to build the environment of many machines for large-scale graphs. In contrast, Seraph, as a fully-external system, exhibits optimal scalability by decoupling the capability of large-scale graph computation from the single machine's memory capacity. Thus, Seraph is low-cost for computing any large-scale graph with a constant memory amount.

6 Conclusion

This work develops a new fully-external graph computation system, Seraph, based on the principle of on-demand processing to save I/O. To purse a higher performance improvement, three practical designs are proposed based on the framework of on-demand processing. Specifically, hybrid format is introduced to store the graph while optimizing graph computation, vertex passing is presented for handling vertex updates efficiently, and selective pre-computation explores the possibility of re-using I/O. Seraph, as a fully-external system, exhibits optimal scalability and offers decent performance. It significantly outperforms the other state-of-the-art fully-external systems on all evaluated graphs, based on our experiments.

Acknowledgments

We sincerely thank our shepherd, Ashvin Goel, 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. CUHK14208521).

References

[1] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 125–137, Santa Clara, CA, July 2017. USENIX Association.

[2] Scott Beamer, Krste Asanovic, and David Patterson. Direction-optimizing breadth-first search. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2012.

[3] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution 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 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[4] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[5] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[6] Wei Chen and Shang-Hua Teng. Interplay between social influence and network centrality: A comparative study on shapley centrality and single-node-influence centrality. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 967–976, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.

[7] Eu2015 dataset from WebGraph. `https://law.di.unimi.it/webdata/eu-2015/` , 2015.

[8] Gsh2015 dataset from WebGraph. `http://law.di.unimi.it/webdata/gsh-2015/` , 2015.

[9] Twitter dataset from WebGraph. `http://law.di.unimi.it/webdata/twitter-2010/` , 2010.

[10] Devdatt Dubhashi, Alessandro Mei, Alessandro Panconesi, Jaikumar Radhakrishnan, and Aravind Srinivasan. Fast distributed algorithms for (weakly) connected dominating sets and linear-size skeletons. *Journal of Computer and System Sciences*, 71, 03 2003.

[11] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. Large-scale graph processing on emerging storage devices. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, page 309–316, USA, 2019. USENIX Association.

[12] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *SIGCOMM Comput. Commun. Rev.*, 29(4):251–262, aug 1999.

[13] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, October 2012. USENIX Association.

[14] Taher H. Haveliwala. Topic-sensitive pagerank. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, page 517–526, New York, NY, USA, 2002. Association for Computing Machinery.

[15] John Hopcroft and Robert Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. Communications of the ACM, 16(6):372–378, 1973.

[16] H. Jeong, S. P. Mason, A.-L. Barabási, and Z. N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411(6833):41–42, May 2001.

[17] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. Grafboost: Using accelerated flash storage for external graph analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 411–424. IEEE Press, 2018.

[18] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT '15, pages 39–50, 2015.

[19] Jooho Kim and Makarand Hastak. Social network analysis: Characteristics of online social networks after a disaster. *International Journal of Information Management*, 38(1):86–96, 2018.

[20] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, October 2012. USENIX Association.

[21] Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor Prasanna. Gpop: A scalable cache- and memory-efficient framework for graph processing over parts. *ACM Trans. Parallel Comput.*, 7(1), mar 2020.

[22] Kartik Lakhotia, Rajgopal Kannan, and Viktor Prasanna. Accelerating PageRank using Partition-Centric processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 427–440, Boston, MA, July 2018. USENIX Association.

[23] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[24] Hang Liu, H. Huang, and Yang Hu. ibfs: Concurrent breadth-first search on gpus. pages 403–416, 06 2016.

[25] 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.

[26] Vladimir V. Makarov, Maxim O. Zhuravlev, Anastasija E. Runnova, Pavel Protasov, Vladimir A. Maksimenko, Nikita S. Frolov, Alexander N. Pisarchik, and Alexander E. Hramov. Betweenness centrality in multiplex brain network during mental task evaluation. *Phys. Rev. E*, 98:062413, Dec 2018.

[27] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135–146, New York, NY, USA, 2010. Association for Computing Machinery.

[28] 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*, ISCA '19, page 116–128, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 24(2):288–300, 2013.

[30] Edward F. Moore. The shortest path through a maze. In Proceedings of the International Symposium on the Switching Theory, 1959, pages 285–292.

[31] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. Graphbig: understanding graph computing in the context of industrial solutions. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[32] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 456–471, New York, NY, USA, 2013. Association for Computing Machinery.

[33] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[34] 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*, SOSP '13, page 472–488, New York, NY, USA, 2013. Association for Computing Machinery.

[35] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. Streaming algorithms for k-core decomposition. *Proc. VLDB Endow.*, 6(6):433–444, apr 2013.

[36] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, feb 2013.

[37] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *2015 Data Compression Conference*, pages 403–412, 2015.

[38] Intel Optane 905P SSD. https://www.intel.com/content/www/us/en/products/details/memory-storage/consumer-ssds/optane-ssd-9-series.html.

[39] Samsung 860 EVO SSD. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/860evo/.

[40] Samsung 970 PRO SSD. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/970pro/.

[41] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. The more the merrier: Efficient multi-source graph traversal. *Proc. VLDB Endow.*, 8(4):449–460, dec 2014.

[42] Keval Vora. LUMOS: Dependency-driven disk-based graph processing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 429–442, Renton, WA, July 2019. USENIX Association.

[43] Keval Vora, Rajiv Gupta, and Guoqing Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*, pages 237–251, 2017.

[44] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 507–522, Denver, CO, June 2016. USENIX Association.

[45] Junlong Zhang and Yu Luo. Degree centrality, betweenness centrality, and closeness centrality in social network. In *Proceedings of the 2017 2nd International Conference on Modelling, Simulation and Applied Mathematics (MSAM2017)*, pages 300–303. Atlantis Press, 2017/03.

[46] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. Wonderland: A novel abstraction-based out-of-core graph processing system. *SIGPLAN Not.*, 53(2):608–621, mar 2018.

[47] 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.

[48] Xiaojin Zhu and Zoubin Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical report, 2002.

[49] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-Centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 301–316, Savannah, GA, November 2016. USENIX Association.

[50] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, Santa Clara, CA, July 2015. USENIX Association.